A RAND NOTE

THE ROSIE LANGUAGE REFERENCE MANUAL

J. Fain, D. Gorlin, F. Hayes-Roth,
S. Rosenschein, H. Sowizral, D. Waterman

December 1981

N-1647-ARPA

Prepared For

The Defense Advanced Research Projects Agency

A ROSIE™ Report

Rand
SANTA MONICA, CA. 90406

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 1981		**00-00-1981 to 00-00-1981**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The Rosia Langauage Reference Manual	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **RAND Corporation,1776 Main Street, PO Box 2138,Santa Monica,CA,90407-2138**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**158**	

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under ARPA Order No. 3460/3473, Contract No. MDA903-78-C-0029, Information Processing Techniques.

Published by The Rand Corporation

A RAND NOTE

THE ROSIE LANGUAGE REFERENCE MANUAL

J. Fain, D. Gorlin, F. Hayes-Roth,
S. Rosenschein, H. Sowizral, D. Waterman

December 1981

N-1647-ARPA

Prepared For

The Defense Advanced Research Projects Agency

A ROSIE™ Report

Rand
SANTA MONICA, CA. 90406

PREFACE

ROSIE (Rule-Oriented System for Implementing Expertise) is a computer programming environment developed at The Rand Corporation under contract MDA903-78-C-0029 from the Defense Advanced Research Projects Agency. ROSIE has evolved from a succession of projects in artificial intelligence and expert systems. The current version of ROSIE differs significantly from all extant programming languages as well as from its own initial design (described in Rand Note N-1158-1-ARPA, Design of a Rule-Oriented System for Implementing Expertise, May 1979).

ROSIE has been implemented in INTERLISP and is currently supported on the TOPS-20 operating system. Additional information about ROSIE can be found in companion RAND Notes N-1648-ARPA, Rationale and Motivation for ROSIE, November 1981, and N-1646-ARPA, Programming in ROSIE: An Introduction by Means of Examples (forthcoming).

SUMMARY

ROSIE is a programming language and programming system for artificial intelligence (AI) applications. The ROSIE language is a stylized version of English. Our primary design goal for the language has been to achieve exceptional program readability. A second goal has been to support the development of significant applications. ROSIE provides a variety of language and programming environment features aimed at this objective. The language allows the programmer to describe complex relationships simply and to manipulate them symbolically and deductively. In addition, it supports network communications and patterned reading and writing to other systems. It also provides for interactive, compiled, and interpreted computing, with a variety of debugging and programming tools.

ROSIE encompasses many of the capabilities of conventional programming languages. It is a general-purpose language offering a variety of typical data types and control constructs found in most high-level languages, together with a few found only in AI languages. Among the features we include are rulesets that generate sets, predicates that test propositions, propositional data types, and some limited forms of deduction.

Using ROSIE, an AI applications programmer can think concretely about the problem domain and translate ideas into a program using substantially the same vocabulary that arises in the English (non-computational) formulation of the model. ROSIE's language will provide a naturalistic style for describing even such mathematical abstractions as graphs, stacks, etc. For example, a user can refer to "a stack whose top element is ...."

This manual consists of a technical discussion of the syntax and semantics of the ROSIE language as well as an explanation of the ROSIE environment as a whole.

CONTENTS

1. INTRODUCTION

1.1. OVERVIEW

This document is a reference manual for users of the ROSIE programming environment. Novice users are referred to Ref. 1 for examples of ROSIE code and to Ref. 2 for an overview of ROSIE design philosophy and motivation.

The ROSIE language has evolved through many phases of design and improvement. As in any language, some limitations in language flexibility and expressiveness still exist and may cause some frustration. This release implements a collection of design decisions complete enough to support the development of a ROSIE user community.

1.2. THE ROSIE ENVIRONMENT

ROSIE attempts to provide a complete working environment for developers of interactive knowledge-based systems. Because such systems are often built around a collection of rules and heuristics for working within a problem domain, ROSIE encourages the user to think in terms of English-like rules and collections of rules, and provides an interactive environment which supports that approach. This support is an integral part of the ROSIE design philosophy.

ROSIE's English-like rule syntax not only allows users to embody knowledge in a convenient and natural form, but improves interaction with others involved in the knowledge acquisition process. A well-written ROSIE program is accessible not only to the programmer, but to the domain experts and others as well. Non-programmers can examine rulesets and suggest modifications directly, become more involved in system development, and ultimately express their knowledge in ROSIE rules. In addition, because programs can easily be scanned and modified interactively, knowledge development becomes practical in a conference or demonstration environment, without the usual delay associated with program modification.

These benefits, however, are not automatic. Although readable ROSIE rules are not hard to generate, it is just as easy to write cryptically as it is to write clearly.

1.3. LIMITATIONS

The purpose of this release is to provide a working version of the language for experimentation and small system development. Many efficiency issues have not been adequately considered, especially concerning space and speed. Compilation of programs is essential due to severe storage limitations on the DECsystem-20* and will greatly improve the speed of running programs. Database access is relatively efficient, but database size is somewhat restricted by the general shortage of available space.

Because we have emphasized language decisions, there is also a general shortage of user conveniences, especially with regard to input/output and data representation. System rulesets allow users to access INTERLISP directly, which requires some knowledge of INTERLISP and a reference manual for that language. Users requiring comprehensive language features should anticipate this.

1.4. NOTATION

Examples in this document follow certain conventions for readability and are always set apart from the text which explains them.

Boldface	Words in boldface are either reserved words or keywords in the language.
Standard	Words in standard font are parameters, examples of items supplied by the user.

Example: **Assert** John **does not** like movies **about** horses.

In addition, the following conventions hold when syntactic constructs are presented or discussed:

Boldface	Words in boldface are fixed parts of language constructs.
Underline	Underlined words are syntactic categories. They represent legal constructs of that type.

---

* Decsystem-20 is a trademark of Digital Equipment Corp., Maynard, Mass.

[,] These characters (brackets) surround optional parts of constructs.

{¦ ¦} These characters (vertical bars and braces) surround a part of the construct which can appear zero or more times in that position.

(¦ ¦) These characters (vertical bars and parentheses) surround a set of alternatives, only one of which can occur in that position. The options are separated by vertical bars.

Examples:

Send [**to** term] **pattern**
action {¦ **and** action ¦}
Open term **to** (¦ **read** ¦ **write** ¦ **append** ¦)

Underscoring is used to highlight new concepts when they are introduced.

1.5. HOW TO USE THIS MANUAL

The ROSIE Reference Manual is designed to be of use to those at all levels of expertise.

New users who are interested primarily in ROSIE as a language (i.e., apart from applications) should examine carefully Appendix A (A ROSIE Primer) first, then concentrate on chapters 2, 3, 4, and 5. In addition, the BNF of the grammar is given in section 7.5. The new user who is oriented toward programming should study Appendixes A and B (Getting Started with ROSIE) before tackling the remainder of the manual.

Users already familiar with ROSIE may have recourse to this manual for a variety of reasons. Those who need a refresher on a particular topic are directed to the Table of Contents or Index for the location of an in-depth discussion of the subject. In general, index entries point toward the location of the major portion of discussion for that topic. A user who merely wants to check for the existence or correct syntax or semantics of a particular action is directed toward one of the three quick references (sections 3.7, 5.14, and 6.7). The BNF (section 7.5) is also a useful source of information for the expert user.

The programmer who is in the debugging phase of a program may find help in one of two locations. First, section 7.2 discusses ROSIE errors in general. As an alternative, the user can turn to the section of the manual discussing the general topic of which the error is an example (e.g., a

syntax error in a relative clause might send you to section 4.5). In many sections, the occurrence of the label "NOTE:" signals a warning about semantics of the particular ROSIE construct that some find counterintuitive.

A final note: In reading this manual cover to cover, the user may notice a fair amount of redundancy; this is intentional. By repeating information in a variety of locations we hope to make it easier for you to find what you need when you need it. In the long run, repetition should facilitate the use of this document as a reference manual.

2. PRIMITIVE STRUCTURES OF THE LANGUAGE

2.1. OVERVIEW

This chapter introduces the basics of any high-level language: the character set (2.2), the primitive data types (2.3), and the reserved words (2.4).

2.2. LEXICAL ANALYSIS

Before the action embodied in a ROSIE sentence can be performed, the English syntax must be parsed into an internal representation that can be executed. The first step in this process is called lexical analysis and involves separating the text into individual units called tokens.

2.2.1. Tokens

The example below shows a typical ROSIE sentence followed by the tokens it contains in uppercase:

Assert <"3,4", 3.4, Dave's age> **is a** tuple **of** elements.
ASSERT < "3,4" , 3.4 , DAVE ' S AGE > IS A TUPLE OF ELEMENTS .

Distinguishing one token from the next is done by using knowledge about three types of characters:

(1) SEPARATOR CHARACTERS: spaces, tabs, and carriage returns.

(2) BREAK CHARACTERS: special characters that the lexical analyzer recognizes as indicating the beginning or end of a ROSIE construct. The break characters are

{ } ` ' " " ( ) , < > ~ = ; :

(3) TERMINAL CHARACTERS: characters that indicate the end of a rule (5.2). The terminal characters are

. ! ? :

Note that terminal characters must be followed by a carriage return to be considered as terminal characters. Thus, the

use of the "." in the number "3.4" above carries its normal meaning. Note also that the ":" is a terminal character only in the special case of ruleset headers (see chapter 6).

The string "3,4" in the example above was not broken apart even though commas are break characters, because strings (which begin and end with double quotes) are considered as one token by the parser. This allows strings to contain any character at all, except the double quote character. Actually, ROSIE programs can read and create strings which contain double quotes using patterns (see section 4.9), but these strings will not parse correctly when typed to the top level or when included in filepackage files (see section 6.1).

2.2.2. Comments

Comments can appear anywhere in a program file. They are preceded by a left square bracket "[" and terminated with a right square bracket "]". Comments can also be nested to any depth, meaning that a comment within a comment will parse correctly. Comments are simply discarded by the parser, so they do not occupy storage when rulesets are loaded.

2.3. PRIMARY LANGUAGE ELEMENTS

Elements are the primitive data types in ROSIE. Rulesets (5.3) can be written to accept elements as parameters and to generate a sequence of elements on request. Elements can also be stored as variable values within rules (5.2). Database entries (3.1) are relationships among elements that can be manipulated by a number of actions.

There are a number of different element types available, each of which represents a different kind of information. Together, they allow programs to manipulate data in many forms.

2.3.1. Evaluation Names

Elements are usually created when terms are evaluated. When an element is printed, the evaluation name (in uppercase) of the element is used. The evaluation name is the string of characters representing an element. Every element type has its own format for creating evaluation names so that the evaluation name will resemble the form of the original term. For example, the term

<The general, 3 + 4, john's mother, john>

is the term commonly used to create tuple elements. The element this term creates when evaluated might print as follows:

<GENERAL SMITH, 7, SARAH LEE, JOHN>

Thus, the evaluation names for the terms "The general", "3 + 4", and "john's mother" are "GENERAL SMITH", "7", and "SARAH LEE" respectively. Not all terms will appear to evaluate, however; the evaluation name for the term "john" is simply "JOHN."

NOTE: Evaluation names always appear in the database in uppercase.

2.3.2. Names

Name elements represent literal names which can contain one or more words. These elements print with spaces separating the individual words in the name. Legal names cannot include words which can be interpreted as numbers or strings.

Examples:
John
Captain Kirk
Ship #3
Employee 566-96-9990A
Washington State University

Names are considered equivalent or equal only when their evaluation names are the same.

2.3.3. Strings

String elements represent strings of any number of characters. They are printed between double quotes. Characters in strings are not converted to uppercase by the parser.

Examples:
""
"THIS IS A STRING"
"Please respond now:"
"566-96-9990A"

Two strings are considered equivalent or equal when their evaluation names are the same.

2.3.4. Numbers

Number elements represent numeric values. They can also include units or labels associated with those values. The ROSIE arithmetic operators (+ - / * **) and the comparison operators (= > >= < <= ~=) and their English equivalents (is equal to, is greater than, is greater than or equal to, is less than, is less than or equal to, is not equal to) will handle and combine units and labels correctly. They will also ensure that operations and comparisons are sensible. For example, attempting to add 33 APPLES to 44 ORANGES will cause an error because the units do not match. Unlabeled numbers, however, can be multiplied and divided by unit elements or label elements; the result contains the correct units or label.

There are two types of number elements, called unit constants and label constants.

The simplest kind of unit constant is just a numeric value, which can be an integer, floating-point, or octal number with optional positive or negative exponents.

Examples:	335	(Integer)
	-25	(Negative integer)
	4.603	(Floating-point)
	3.2E10	(Floating-point with positive exponent)
	45E-10	(Floating-point with negative exponent)
	7742Q	(Octal)

Unit constants can also have units as in the following examples:

Examples:
```
33 Oranges
24 miles/hour
-4.7 feet/second**2
800 feet*pounds/seconds**2
200 metric tons/cubic feet
13.7 1/feet**2
13.2 feet**-2
```

In general, units consist of a numerator and an optional denominator. As seen above, the evaluation name of a unit element separates the numerator from the denominator with the "/" character, precedes exponents by "**", and separates units multiplied together with "*". All units following the

first "/" are assumed to be part of the unit denominator, unless they have negative exponents.

Label constants are numeric values which are preceded by one or more labeling words. These are examples of label constants:

Examples: Probability .33
Certainty 7
Ground Combat Division 13

Unit and label constants exist to give the user more flexibility in representing numeric values and are supported by the built-in arithmetic and comparison operators (see section 4.3.6).

Two numbers are equal if their numeric values are equivalent and both are either unit or label constants with equivalent units or labels.

2.3.5. Tuples

Tuple elements represent ordered lists of elements. The evaluation name for a tuple element consists of a left angle bracket "<", followed by the evaluation names of each element in the tuple separated by commas, and terminated by a right angle bracket ">".

Examples: <>
<1, 2, 3>
<John, 33.5, "A string", <1, 2, 3>>

Two tuples are equal only if both tuples contain equal elements in identical order.

2.3.6. Other Primary Language Elements

Another type of primitive data structure is the proposition. However, because of its close relation to the primitive sentence, discussion of this element type is deferred to section 4.6.2.

Similarly, class elements are considered primitive in ROSIE. They are discussed in section 3.6 because of their close relation to the database.

2.4. RESERVED WORDS

Reserved words are tokens which can only be used in specific syntactic constructs. All break characters and terminal characters are reserved words. These are the reserved words in ROSIE:

```
{ } ` ' ( ) , < > ~ = @ ; + - * ** / : . ? !

about    above    across   after    against  along
among    around   as       at       because  before
behind   below    beside   by       during   for
from     in       inside   into     near     of
on       onto     outside  over     since    through
to       under    until    up       while    with
within   without  toward   unless   per

and      or       there    not      a        an
was      is       will     did      does     provably
be

matched  has      greater  less     equal    otherwise

the      that     new      some     every    each
any      such     one

who      which    whom     where    whose

let      assert   deny
```

3. STORING SENTENCES: THE DATABASE

3.1. OVERVIEW

A thorough understanding of Rosie's semantic constructs requires knowledge of ROSIE's data storage mechanisms. The ROSIE programming environment consists of a database and a number of rulesets (programs) written to manipulate this database. ROSIE can store data via temporary bindings or in a database. Temporary bindings store data produced within a rule for use only within that rule (see 5.2). The database, on the other hand, can store any data for any length of time. In order to write reasonable ROSIE programs it is essential to have an intimate knowledge of the database, its structure, its permissible relations, and its elemental units.

The ROSIE database consists of two conceptually separate layers. The first is the "physical" database. It contains the affirmed propositions, i.e., those relationships that have been explicitly added to the database through the assertion of a proposition or a sentence. The second is the "virtual" database. The "virtual" database consists of those relationships that can be computed, via reasoning algorithms, from other relationships in the database. Relations in the virtual database are stored as relations in the physical database among sets of elements rather than individual elements. The element set is defined by a class element (see 3.6). A relation whose object is a class element implicitly specifies a number of virtual relations each of which relates an element from the set described by the class element. These virtual relationships do not exist in the database; instead they are computed each time they are needed.

ROSIE's "layered" database can be understood through a simple example. Assume that the database contains the following sentences (the relation shows how ROSIE represents the sentence internally):

Asserted Sentence	Relation
John **is** mortal	is-mortal(John)
John **is a** minister	is-a-minister(John)
Fred **is a** man	is-a-man(Fred)
Any man **is** mortal	is-mortal(**any** man)

All four of these relations are in the physical database. The relationships "is-mortal(John)" and "is-a-minister(John)" state that the element "John" satisfies "is-mortal" and "is-a-minister". The third

relationship "is-a-man(Fred)" declares that the element "Fred" satisfies "is-a-man." In a similar manner, the fourth relationship "is-mortal(**Any** man)" implies that the element "**any** man" satisfies "is-mortal." Because of this relationship, "**any** man" (a class element) will automatically insure that every element in the database that satisfies "is-a-man" also satisfies "is-mortal." Thus the relationship "is-mortal(Fred)" exists, but only in the virtual database. Now, in order to deduce this virtual relationship, ROSIE needs to compute the relation by performing a search over all relevant relationships in the database. In general, a set of relationships stored explicitly in the database requires more memory for their representation than the same set of relationships stored virtually, while a set of relationships stored virtually requires more computation for their retrieval than the same set of relationships stored explicitly.

Relations cannot exist in a vacuum; they must refer to something. The objects that ROSIE relationships reference are precisely those elements discussed in section 2.3 and the class elements discussed in section 3.6.

The relational scheme of representation provides a tool for writing readable and intuitive programs, but not all types of information are easily represented with relationships. ROSIE is currently limited in its ability to handle data which, for example, is best represented with multi-dimensional arrays or random-access files. Programs which use such data can resort to system rulesets when ROSIE constructs prove cumbersome.

3.2. RELATIONAL FORMS AND RELATION-NAMES

Relationships which can appear in the database are constructed from the legal relational forms. These forms define the structure of all legal relationships, which are built by filling in a form with a relation-name and one or more elements. These are the relational forms:

Legal Relational Forms:

element (¦ **was**¦**were** ¦) [**not**] a[**n**] relation-name {¦ prep element ¦}
element (¦ **is**¦**am**¦**are** ¦) [**not**] a[**n**] relation-name {¦ prep element ¦}
element **will** [**not**] **be** a[**n**] relation-name {¦ prep element ¦}

element (¦ **was**¦**were** ¦) [**not**] relation-name {¦ prep element ¦}
element (¦ **is**¦**am**¦**are** ¦) [**not**] relation-name {¦ prep element ¦}
element **will** [**not**] **be** relation-name {¦ prep element ¦}

element **did** [**not** relation-name {¦ prep element ¦}
element (¦ **does**¦**do** ¦) [**not**] relation-name {¦ prep element ¦}

element **will** [**not**] relation-name {¦ prep element ¦}

element (¦ **was**¦**were** ¦) [**not**] relation-name element {¦ prep element ¦}
element (¦ **is**¦**am**¦**are** ¦) [**not**] relation-name element {¦ prep element ¦}
element **will** [**not**] **be** relation-name element {¦ prep element ¦}

element **did** [**not**] relation-name element {¦ prep element ¦}
element (¦ **does**¦**do** ¦) [**not**] relation-name element {¦ prep element ¦}
element **will** [**not**] relation-name element {¦ prep element ¦}

Examples of legal relationships:

Mr Jones **was a** president **during the crisis.**
Mr Smith **is a** president **in** 1981.
Mr Lewis **will be a** president **in** 1985 **for** 1 year.

Mr Jones **was** corrupt **before** 1900.
Mr Smith **is not** corrupt.
Mr Lewis **will be** corrupt **after** elected.

Mr Jones **did** conspire.
Mr Smith **does** aspire **to** greatness.
Mr Lewis **will not** inspire **for** obvious reasons.

Mr Jones **was** prolonging taxes **until** November.
Mr Smith **is** planning miracles.
Mr Lewis **will not be** impeached easily.

Mr Jones **did** cause controversy **in** office.
Mr Smith **is** causing concern **among** liberals.
Mr Lewis **will** inspire fear **among** sane persons.

In the above examples, "president," "corrupt," "conspire," "aspire," "inspire," "prolonging," "planning," "impeached," "cause," "causing" and "inspire" are all relation-names. They are not considered elements, but rather are part of the relational forms that embed them. In section 5.3 we explain how relation-names can be used to instruct database access operations to invoke a ruleset for answers instead of interrogating the global database.

NOTE: ROSIE does not "understand" the difference between singular and plural nouns. ROSIE does not recognize "captains are" as the plural of "captain is." "Are," "am," "were," and "do" are available to the user to make code more English-like.

3.3. THE **IS A** RELATION

The "element **is a** relation-name {¦ preposition element ¦}" form (along with its negation) is perhaps the most important of the forms. The **is a** relation defines class memberships. Thus, by asserting "John is a male," we assert that John belongs to the class of males. This differs from "John is male," which merely gives the name "John" the attribute "male."

Class relations are at the heart of the ROSIE database. The **is a** relation supports the use of **the** and of the **let** action. In ROSIE, when you refer to "the male" you are referring to an element that satisfies the is-a-male relation. If no such element exists, reference to it produces an error. If more than one such element is present in the database, one is chosen arbitrarily. However, by saying "Let the male be John" you can rid the database of all elements satisfying the is-a-male relation and install John as the only such element. Thus, **let** and **the** can work in tandem to guarantee the existence of a unique element satisfying some relation. The use of the determiner **a** has an effect very different from that of **the**. Referring to "a man" causes ROSIE to first check the database to see if any element satisfying the is-a-man relation exists. If so, the one it finds is the man it assumes reference is being made to. But if no element exists in the class of men, ROSIE will create one and add a sentence like "MAN #1 is a man" to the database. Man #1 will then be the man referred to in the remainder of the rule.

Unlike the other relational forms, the **is a** relation is special because it can be used not only when testing the database but also when generating set members. The ROSIE rules in which this occurs are discussed in section 3.6.

One last remark about the exceptional nature of **is a** should be made. When a prepositional phrase is present in an **is a** relation we can distinguish between the relation itself and the effect of the relation on testing and generation. If, for example, we used the sentence "John is a man on Mars," the relation would be is-a-man-on. But the relation is dyadic, i.e., has two arguments--John and Mars. Another way of thinking about this is that while the relation is is-a-man-on, John belongs to the class of men on Mars. If we know that "John is a man on Mars" and "Steve is a man on Neptune" there need not be any confusion during testing. John is-a-man-on and Steve is-a-man-on, but John is a member of the class of men on Mars and Steve is a member of the class of men on Neptune. Thus, if we say, "If there is a man on Mars, display that man," John would be displayed. Similarly, if we are looking for a man on Neptune, Steve would be displayed. In both cases the relation is is-a-man-on, but the element following the preposition determines which person we meant by determining the class. Since the preposition forms part of the relation, the anaphoric reference is resolved in "display **that** man" without the need to reuse the entire prepositional phrase.

NOTE: When a prepositional phrase acts to modify a relation, ROSIE does not assume the simpler relation also holds. Thus, "John is a man on Mars" does not entail "John is a man."

3.4. NEGATION AND TRUTH VALUES

Databases are automatically kept consistent with regard to negation. The negation of a relationship is defined as that relationship which is identical except for the appearance or absence of the word **not**. So, for example, "John is a man" and "John is not a man" cannot both appear in the same database. When one is added (using **assert**, **let**, or other database actions), the other is automatically discarded.

In ROSIE, the absence of a relationship does not imply that the negation of that relationship is true. A three-valued logic has been adopted which allows three states for a given relationship: provably true, provably false, and undecided. The following ROSIE session illustrates this.

Sample session:

```
                [ ROSIE  Sunday, February 1, 1981  4:33pm ]

<1> Clear database.
<2> ?
[ Global Database ]

<3> If John is a man, display YES.
<4> If John is not a man, display NO.
<5> Assert John is a man.
<6> ?
[ Global Database ]
    JOHN is a man.

<7> If John is a man, display YES.
YES
<8> If John is not a man, display NO.
<9> Assert John is not a man.
<10> ?
[ Global Database ]
    JOHN is not a man.

<11> If John is a man, display YES.
<12> If John is not a man, display NO.
NO
<13> Logout.
```

NOTE: The statement that the database is kept consistent with respect to negation should be qualified. ROSIE's ability to check for inconsistencies is limited to immediately comparable relations and does not include reasoning through and reconciling the implications of a set of database sentences as in the following example:

Assert John **does** like **any** girl.
Assert John **does not** like Mary.
Assert each of Mary, Jane **and** Beth **is a** girl.

NOTE: Sentences of the form "element **is not a** description" (e.g., "John is not a bad boy") are stored as negations of the components (i.e., "John is not a boy" and "John is not bad").

3.5. GLOBAL, PRIVATE, AND ALTERNATE DATABASES

ROSIE programs usually manipulate relationships in databases as their primitive units of information. There is one database called the global database which usually contains the main body of information required by a program. All rulesets can interrogate and modify the global database.

In addition, each ruleset invocation allocates a "private database." The relations asserted into a private database are discarded at a ruleset's termination, and so are useful for storing temporary relationships that will not be needed outside of the ruleset. Section 5.8 explains this more fully.

Finally, ROSIE permits you to create alternate databases. These databases act just like the global database but segregate information from one another and from the global database. To create a database, use the action "**activate** term," where term must evaluate to a name. Using the action **deactivate** or just **activate** by itself always returns you to the global database. Once you are "in" an alternate database (i.e., you have activated it), any assertions, denials, lets, database testing, etc., pertain to that database. Because rulesets do not reside in a database, these are accessible from all databases. The system-defined procedures "add PROP to DATABASE" and "remove PROP from DATABASE" allow you to assert and delete primitive sentences in one database while working in another. The system-defined procedure "show DATABASE" allows you to see the contents of any database while residing in any other.

3.6. CLASS ELEMENTS

One concept which appears often in real-world knowledge is that of groups or classes of entities. ROSIE supports the concept of class with a number of unique features. The most basic of these is the ability to enumerate all elements within a database that conform to a particular constraint. In the following session, the use of **each of** and **every** demonstrates this feature.

Classes are even more useful because of the special interpretation given to class elements. A class element consists of the reserved word **any** followed by a description. The class element itself represents every database element that satisfies that description. Thus, a class element designates a variable number of elements and the actual elements generated depend upon the description. A database relationship which involves class elements is treated exactly as though a separate relationship existed for all elements described by the class. In the following session the terms "**any** girl," "**any** man," and "**any** person" demonstrate the use of class elements.

Sample session:

```
            [ ROSIE  Sunday, February 1, 1981  5:27pm ]

<1> Assert each of John, Dick, Dave and Sam is a man.
<2> Assert each of Mary, Jane, Sue and Sally is a woman.
<3> Assert any man does like any woman.
<4> Assert any woman is a person and any man is a person.
<5> ?
[ Global Database ]
    ANY MAN does like ANY WOMAN.
    JOHN is a man.
    DICK is a man.
    DAVE is a man.
    SAM is a man.
    MARY is a woman.
    JANE is a woman.
    SUE is a woman.
    SALLY is a woman.
    ANY WOMAN is a person.
    ANY MAN is a person.

<6> If John does like Dick, display YES.
<7> If John does like Sally, display YES.
YES
<8> Display every man.
JOHN
DICK
```

```
DAVE
SAM
<9> Display every person.
MARY
JANE
SUE
SALLY
JOHN
DICK
DAVE
SAM
<10> Forget about every man.
<11> ?
[ Global Database ]
    ANY MAN does like ANY WOMAN.
    MARY is a woman.
    JANE is a woman.
    SUE is a woman.
    SALLY is a woman.
    ANY WOMAN is a person.
    ANY MAN is a person.

<12> Display every person.
MARY
JANE
SUE
SALLY
<13> logout.
```

In statements <1> and <2> eight relationships are asserted using two **is a** relations. In <8> the user asks ROSIE to enumerate all men by using **every**. ROSIE scans the global database looking for the appropriate **is a** relationship and generating the elements which are members of the appropriate class.

The example above points out an important use of class elements--delayed evaluation. If we had said in <4> "assert every woman is a person and every man is a person," the sentences "JOHN is a person," "DICK is a person," etc., would have been added to the database. Since the class element acts as a placeholder for all the individual elements of a class, the effect of using the **any** construction is to delay the enumeration of all the individual **is a** relations until they are needed. In short, the use of class elements enables you to store more knowledge in the database using less space than if you stored the same knowledge explicitly.

A class element is different from other elements in that it is equal only to elements which are members of the class it describes, rather than to

identical elements. In addition, two class elements are equal only if an element exists which is a member of both classes.

3.7. DATABASE COMMANDS: QUICK REFERENCE

These actions provide access to the current database:

assert sentence {¦ **and** sentence ¦} -- adds relationships to the current database. Each sentence is interpreted as described in section 4.8.2.

deny sentence {¦ **and** sentence ¦} -- removes relationships from the current database. Each sentence is interpreted as described in section 4.8.2.

assert term {¦ **and** term ¦} -- each term must evaluate to a proposition element. Adds each proposition to the current database. Equivalent to: **assert** term **is provably true** {¦ **and** term **is provably true** ¦}. (see 4.6.4)

deny term {¦ **and** term ¦} -- each term must evaluate to a proposition element. Removes each proposition from the current database. Equivalent to: **deny** term **is provably true** {¦ **and** term **is provably true** ¦}. (see 4.6.4)

let definite-description **be** term {¦ **and** definite-description **be** term ¦} -- the definite-description must be either "**the** description" or term'**s** description." The description from the definite-description is extracted and used to **deny** every "**is a** description" and **assert** "term **is a** description." **Let** leaves the terms as the only element that fits the description. For example, these rules are equivalent and leave John Smith as the only mayor of Chicago known to the database:

> **Let the** mayor **of** Chicago **be** John Smith.
> **Let** Chicago'**s** mayor **be** John Smith.

Let permits the order of the definite-description and the term to be reversed if no confusion arises. However, if the term itself begins with **the**, ROSIE assumes the definite-description precedes **be** and the term follows **be**. Thus, the following are two additional ways of performing the previous action.

Let John Smith **be the** mayor **of** Chicago.
Let Chicago's mayor **be** John Smith.

create a[n] description -- fabricates a new name element and asserts element is a description. **Create** builds the name from the relation-name given in the description and appends #n where n is an integer which makes the name unique. Useful for applications where the elements created represent objects which the user does not wish to name explicitly. The #n is generated in sequence starting with #1. ROSIE will insure its own generated numbers are sequenced properly but will not check to see if the user has defined identical names explicitly.

forget about term -- removes all relationships from the current database that explicitly contain the element value of term.

describe term -- displays every database sentence containing the term.

term ? -- equivalent to **describe** term.

? -- displays the statements in the current database.

clear database -- removes all relationships from the current database only.

activate [term] -- sets the current database to term. If term is omitted the global database is selected.

deactivate -- sets the current database to be the global database.

database -- a system generator that produces the name of the current database.

databases -- a system generator that produces a tuple containing the name of each database that has been activated.

PROPOSITION **is true in** DATABASE -- system-defined predicate that tests the truth of the proposition in the given database. Concludes true or false.

add PROPOSITION **to** DATABASE -- system-defined procedure that asserts the proposition into the specified database.

remove PROPOSITION **from** DATABASE -- system-defined procedure that denies the proposition in the specified database.

show DATABASE -- system-defined procedure that is equivalent to ? for the specified DATABASE.

dump as term -- term must evaluate to a single-word name element. Saves the entire global database on the file name.DATABASE in machine-readable format. The database is not changed. Dump is useful for creating and editing databases using the ROSIE interactive support features.

restore term -- term must evaluate to a single-word name element. Reads a file name.DATABASE which was created by the **dump** action and restores the global database to its state when the database file was created. The previous contents of the global database are lost.

4. THE SYNTAX OF THE ROSIE SENTENCE

4.1. OVERVIEW

This section describes how the primary elements and reserved words combine to form higher-level ROSIE constructs: descriptions (4.2), terms (4.3), and verb phrases (4.4). These constructs, in turn, combine to produce the different types of ROSIE sentence (4.6). Finally, section 4.9 discusses patterns and pattern matching--a mechanism for extracting specific information from an existing sentence.

4.2. DESCRIPTIONS

A ROSIE description represents a class of elements. They can be used within terms (4.3), conditions (4.7), and actions (4.8) to test for class membership or search for all the members of a class in the database. The classes represented are implicit; that is, descriptions describe the elements they represent rather than enumerate them explicitly. The set of elements which a description represents changes as the database changes. Thus ROSIE descriptions act very much like natural-language descriptions and are designed to closely resemble such descriptions in English.

Descriptions are used by ROSIE constructs in three ways:

1. Testing an element for class membership, which simply asks whether a description fits that element.

2. Generating some or all of the elements in a class. This is a fancy database lookup which scans the database for elements that fit the description.

3. Adding or removing an element from a class. This is accomplished by adding or removing all the **is a** relationships in the database that make that element a member of the class. The **is a** relations are extracted from the description.

Thus, using this sample description:

full-time employee **of** VitaTech **who does** play tennis

we can write three rules that illustrate the use of descriptions.

Sample rules:

```
If John Doe is a full-time employee of VitaTech
       who does play tennis,
       go tell that employee about the competition.

Display every full-time employee of VitaTech who does
     play tennis.

Assert John Doe is a full-time employee of VitaTech
       who does play tennis.
```

A description can represent one or more primitive relationships to be tested or asserted about. For example, the third rule above will add these relationships to the database when executed:

```
John Doe is full-time
John Doe is an employee of VitaTech
John Doe does play tennis
```

Similarly, the first rule checks to see if these three primitive relationships exist in the database. The second rule generates the set of elements x, such that x is full-time, x is an employee of VitaTech, and x does play tennis.

Descriptions are sometimes used to reduce the number of rules needed to express a concept. For example, the third rule above could have been written as follows:

```
Assert John Doe is full-time
   and John Doe is an employee of VitaTech
   and John Doe does play tennis.
```

NOTE: An important point to remember is that ROSIE distinguishes between descriptions and compound names purely by syntactic means. For example, in "Tom Brown is young," "Tom Brown" is a name, while in "The large ship is burning," "large ship" is a description, since it is preceded by **the**. ROSIE would be just as happy with "Large ship is young" and "The Tom Brown is burning" in which case "Large ship" would be considered a compound name and "Brown" a term modified by the adjective "Tom."

4.2.1. Simple Descriptions

The simplest description represents the class of elements where the primitive relationship

element **is a** relation-name {¦ preposition element ¦}

exists in the database. Descriptions of this form represent all the elements where the implied relationship has been asserted. The following are examples of simple descriptions and the relationships they represent.

Sample descriptions:

company
employee **of** VitaTech
candidate **for** public office **in** 1981

Relationships represented:

element **is a** company
element **is an** employee **of** VitaTech
element **is a** candidate **for** public office in 1981

Adjectives and relative clauses can be used to further restrict the represented class of elements.

NOTE: Sentences of the form "element **is not a** description" (e.g., "John is not a bad boy") are stored as negations of the components (i.e., "John is not a boy" and "John is not bad").

4.2.2. Descriptions with Relative Clauses

Relative clauses place additional restrictions on the class represented by a description. A number of relative clause forms can be used. Each form restricts the class by specifying additional conditions that the element must satisfy.

Relative clauses follow the descriptions they modify. The following are examples of descriptions with relative clauses and the relationships which they represent.

Sample descriptions:

company **which is** bankrupt

employee **of** VitaTech **who does** play tennis **and who did** retire

candidate **for** public office **in** 1981
who did serve **in** Congress
and who President Brown **does** endorse

Relationships represented:

element **is a** company
element **is** bankrupt

element **is an** employee **of** VitaTech
element **does** play tennis
element **did** retire

element **is a** candidate **for** public office **in** 1981
element **did** serve **in** Congress
President Brown **does** endorse element

The legal relative clause forms are described and examples of their use are given in section 4.5.

4.2.3. Descriptions with Adjectives

Adjectives are the simplest way to add conditions to a description. Each adjective implies an additional relationship of the form

element **is** adjective {¦ preposition element ¦}

Adjectives are relation-names which appear immediately before the descriptions they modify. For example, these pairs of descriptions are equivalent:

company **which is** bankrupt
bankrupt company

employee **who is** retired
retired employee

candidate from Colorado **who is** popular **and who is** Republican
popular Republican candidate

4.2.4. Description Variables

Description variables are local variables used within a description. They are bound to the element being generated or tested by the description. Every description, when evaluated, binds a variable associated with it. This variable can be referenced either explicitly (using the variable's name) or implicitly (using anaphora--the term "**that** relation-name"--or through one of the relative clause forms). The description variable is usually invisible to the user and referenced implicitly. In some cases, however, implicit reference is not adequate. In order to disambiguate, the user must supply a variable name explicitly and then use the variable when needed. These are some descriptions which use explicit description variables:

```
integer (i) such that i is greater than 300

American citizen (c)
        such that c was charged with some crime
        and where c was acquitted of that crime

popular Republican candidate (c)
        where c does hold some public office
        and where President Brown does endorse c
```

The following sets of rules are equivalent. Each set demonstrates the explicit use of a description variable along with two methods that reference the variable implicitly.

```
Display every man (m) such that m does support ERA.
Display every man where that man does support ERA.
Display every man who does support ERA.

Display every teacher (t) such that
        John is a student of t.
Display every teacher where
        John is a student of that teacher.
Display every teacher of whom John is a student.

Display every mother (m) such that
        there is more than one child of m.
Display every mother such that
        there is more than one child of that mother.
Display every mother who has more than one child.
```

The following rule demonstrates a situation where explicit variables are required:

```
For each integer (x) from 1 to 100,
        if there is an integer (y) from 1 to 200
                such that
                (y is a prime and x ** y < 34E10),
        display y.
```

Description variables, whether used implicitly or explicitly, are always bound to the elements they generate, test, or make assertions about. This works for descriptions used in any construct. For example, the following rules refer implicitly to description variables which are bound to the appropriate elements:

```
For each father of any child,
        display that father and display that child.

If John is a tall man, display that man.

Assert John is an exemplary student
and VitaTech will hire that student after graduation.
```

NOTE: It is important to remember that the use of a variable as shown in the examples above holds only over a single rule. Thus

```
Display every person(p) such that p is a man
        and assert p is a male.
```

will have the desired effect, but

```
Display every person(p) such that p is a man.
Assert p is a male.
```

will cause the sentence "p is a male" to be added to the database.

4.2.5. Descriptions as Generators

Many constructs use descriptions to generate elements. For example, these rules all use the description "citizen of Los Angeles" to generate one or more elements:

Display some citizen **of** Los Angeles.
Display every citizen **of** Los Angeles.
For each citizen **of** Los Angeles,
go supply **that** citizen **with** oxygen.

Elements are generated from descriptions by searching the database for relationships which assert some element "**is a** description." The above rules all search for relationships of the form

element **is a** citizen **of** Los Angeles

A description with relative clauses and adjectives will make the tests implied by those restrictions for each element it generates. For example, the following rule finds "citizen of Los Angeles" in the database and tests to see if that element is wealthy and if that element will support Reagan:

Display every wealthy citizen **of** Los Angeles
who will support Reagan.

Since descriptions generate elements from the database, they can also generate from generator rulesets (5.5) when appropriate. If the generator "citizen of city," for example, is defined, the above rule would invoke the generator to produce elements after scanning the database. The relationships "element **is** wealthy" and "element **will** support Reagan" will be tested for each element before it is generated. Note also that the predicate "citizen will support candidate" might be invoked in addition to scanning the database if such a predicate ruleset (5.6) were defined.

4.2.6. Descriptions as Class Membership Tests

Descriptions in conjunction with other forms can act as implicit conditionals to test each element against a particular set of constraints. The following rules illustrate this usage:

If the student **is a** healthy adult male,
go enlist **that** student.

If the mayor **is a** wealthy citizen **of** Los Angeles,
go ask **that** mayor **for** financial support.
Display every candidate
who is a Republican proponent **of** ERA.

Relationships tested:

element **is** healthy
element **is** adult
element **is a** male

element **is** wealthy
element **is a** citizen **of** Los Angeles

element **is** Republican
element **is** a proponent **of** ERA

4.2.7. Descriptions and Modification

The **assert**, **deny**, and **let** actions can all be used to make multiple changes to the database by permitting a description to be the object of an **is a** relation. Such a "sentence" represents a number of relationships and the assertion, denial, or reassignment via **let** of such a sentence will assert, deny, or reassign each of the implied relations. This sample session demonstrates the convenience of these features.

Sample session:

[ROSIE Friday, February 6, 1981 4:19pm]

```
<1> Assert Mary Jones is a young girl who will become famous.
<2> ?
[ Global Database ]
    MARY JONES will become FAMOUS.
    MARY JONES is a girl.
    MARY JONES is young.

<3> Deny Mary Jones is a young girl.
<4> ?
[ Global Database ]
    MARY JONES will become FAMOUS.

<5> Deny Mary Jones will become famous.
<6> Assert John Doe is a male employee of VitaTech
    whose salary is large and who does play tennis on weekends.
<7> ?
```

```
[ Global Database ]
    JOHN DOE does play TENNIS on WEEKENDS.
    JOHN DOE is an employee of VITATECH.
    LARGE is a salary of JOHN DOE.
    JOHN DOE is male.

<8> Clear database.
<9> Let Max Schneider be the exemplary student of music at Hoover High
    who will compete in international competition.
<10> ?
[ Global Database ]
    MAX SCHNEIDER is exemplary.
    MAX SCHNEIDER is a student of MUSIC at HOOVER HIGH.
    MAX SCHNEIDER will compete in INTERNATIONAL COMPETITION.

<8> Logout.
```

4.2.8. Anaphoric Descriptions

Sometimes a complex description must appear more than once in a rule. Rather than requiring the repetition of the full description in the second instance, ROSIE permits the use of an anaphoric reference. A **such** followed by the description's relation-name is an appropriate form. The following rules show how an anaphoric reference can be used:

```
If there is an exemplary student of mathematics
  who will graduate before 1984,
  go recruit (every exemplary student of mathematics
  who will graduate before 1984) for summer employment.
```

```
If there is an exemplary student of mathematics
  who will graduate before 1984,
  go recruit every such student for summer employment.
```

NOTE: The resolution of anaphora always refers to an element named in a preceding description. Thus,

```
Assert gs-level 13 is a group
       and let the salary-range of that group be $12K-16K.
```

results in the error "no referent for THAT." In the example, **that** tries to refer to "group" but the only usage of "group" does not occur inside a description. A way of expressing this sentence correctly is

Assert gs-level 13 **is a** group **whose** salary-range
is $12K-16K.

4.3. TERMS

Terms are the syntactic constructs that represent elements. Terms become elements when evaluated within the context of the rule that contains them. For example, the terms

"a string"
4.0 + 5.3
<3 + 2, 4 - 1>
The mayor **of** Los Angeles

when evaluated become the elements

"a string"
9.3
<5, 3>
Tom Bradley

Most terms evaluate to a single element. That element is then used by the construct in which the term appears.

4.3.1. Name Terms

Name terms evaluate to name elements: one or more words separated by blanks.

Examples: John
John Wesley Harding
Ship #2

4.3.2. String Terms

String terms are characters enclosed in double quotes. They evaluate to string elements.

Examples: ""
"Please enter command:"
"566-96-9990"

4.3.3. Number Terms

Number terms evaluate to number elements. These terms cannot be distinguished from the number elements to which they evaluate (see 2.3.4).

Examples:

```
335              (Integer)
-25              (Negative integer)
4.603            (Floating-point)
3.2E10           (Floating-point with positive exponent)
45E-10           (Floating-point with negative exponent)
7742Q            (Octal)

33 Oranges
24 miles/hour
-4.7 feet/second**2
800 feet*pounds/seconds**2
200 metric tons/cubic feet
13.7 1/feet**2

Probability .33
Certainty 7
Ground Combat Division 13
```

4.3.4. Tuples as Terms

Tuple terms evaluate to tuple elements. Tuples consist of any number of terms enclosed by angle brackets and separated by commas. Each term within the tuple is evaluated when the tuple is evaluated.

Examples:

```
<>
<John, Bill, Sue>
<3 + 4, "a string", The mayor of Los Angeles>
```

when evaluated, these become

```
<>
<JOHN, BILL, SUE>
<7, "a string", TOM BRADLEY>
```

4.3.5. Variables as Terms

Variable terms evaluate to the element bound to the variable. A variable term can be an explicit variable name or an implicit variable reference through anaphoric resolution. The term "**that** relation-name" refers to a preceding description and thus the element that description represents.

Examples:

If there is a person (x) **and** x **is a** man, **display** x.

If there is a person **and that** person **is a** man,
display that person.

4.3.6. Arithmetic Expressions as Terms

Expression terms perform arithmetic operations on number elements. Each term in an expression must evaluate to a number element. In addition, the units or labels which are part of the number elements must be compatible according to the following rules (see 2.3.4 for discussion of label and unit constants).

- When numbers are added or subtracted, either both numbers must be label constants with the same label or both numbers must be unit constants with the same units. The result is a number element with the same label or units.

- When numbers are multiplied or divided, they must either both be label constants with the same label or both be unit constants possibly with different units, or one of the two numbers must be unlabeled and without units. The result is a number element which has the appropriate label or units.

- The exponent operation (**) requires that the exponent be a numeric value without units or labels. The other operand can be any number element. If it is a label constant, the result retains the same label. For unit constants, the units also reflect the exponentiation.

Examples of legal operations:

```
3 + 4
14 apples - 7 apples
The distance from Paris to London * 2
16.3 / 2
certainty .2 * 5
The value ** 2
```

Examples of illegal operations:

```
3 apples + 4 oranges
14 - 7 apples
Certainty .5 * Probability .7
4 ** 15 miles
```

The evaluation of arithmetic expressions proceeds according to standard operator precedence. Exponentiation (**) is evaluated first, grouping right to left; multiplication and division are next, grouping left to right; and addition and subtraction are last, also grouping left to right.

NOTE: It is important to remember that ROSIE has no concept of the semantic meaning of a term. Thus, while "14 apples - 7 apples" results in "7 apples," it is also true that "Girl Scout Troup 6 - Girl Scout Troup 4" results in "Girl Scout Troup 2".

4.3.7. Other Types of Terms

The terms described in this section perform specialized operations on elements. Some create new elements, others access the database. The syntactic form of each is given, followed by a brief explanation and an example.

the description -- this term generates one element from the description. That element is the value of the term. If no element satisfying the description exists in the database, it is an error. These are examples of terms of this form:

The enemy ship
The son **of** John **who is** responsible **for** household chores
The logarithm **of** 14.8

term**'s** description -- this term is shorthand for "**the** description **of** term". For example, these pairs of terms are equivalent:

The father **of** John
John**'s** father

The mayor **of** Chicago **in** 1971
Chicago**'s** mayor **in** 1971

the (¦ **number** ¦ **string** ¦ **name** ¦) pattern -- this term builds a string from the pattern and converts it to an element of the type indicated. The examples below show a few sample terms and their element values:

The string {"Hi", 1 blank, John Doe}	"Hi JOHN DOE"
The number {37, ".", 3 + 2}	37.5
The name {John, 1 blank, Doe}	JOHN DOE

the tuple containing each description -- this term returns a tuple that contains all the elements generated from the description. This allows classes to be converted into an explicit tuple representation.

a[n] [**new**] description -- this term, without the **new** option, will attempt to generate an element from the description. If no such element exists, a new one is created and the assertion "element **is a** description" is added to the database. If "new" appears, a new element will always be created. In that case, the term works just like the **create** action and returns the new element (see documentation for the **create** action in 3.7). The following rules demonstrate how this term can be used:

> **Display a** solution **to** Problem 3.
> **Assert a new** enemy ship **did** appear **at the** current time.

In the first example, if at least one sentence "element **is a** solution **to** problem 3" exists in the database, the value of the element will be displayed. If no such relation exists in the database, this action will result in the addition of "SOLUTION #1 is a solution to PROBLEM 3" to the database.

4.3.8. Pseudo-terms: some, every, each of, one of

The pseudo-terms provide a unique way of scanning a group of elements to perform an action or test a condition. For the "**some** description" and "**every** description" terms, the elements scanned are members of a class generated from a description. For the **each of** and **one of** terms, the list of elements to scan is given explicitly. These are the pseudo-term forms:

> **some** description
> **every** description
> **each of** term {¦ , term ¦} [,] **and** term
> **one of** term {¦ , term ¦} [,] **or** term

Pseudo-terms do not evaluate to a single element like other terms. Instead, they change the meaning of the condition or action which contains them. In most cases, the new meaning is identical to what a ROSIE-naive user would expect from reading the rule text. (Pseudo-terms as used in actions work differently from pseudo-terms as used in conditions). The rest of this section explains how psuedo-terms work and gives examples.

An action which contains an **every** or **each of** pseudo-term will be executed once for each element indicated. An action which contains a **some** or **one of** pseudo-term will be executed only once, using only the first element generated (if any). The rules below demonstrate:

```
Go attack every enemy ship.
Go attack each of ship #1, ship #2 and ship #3.
Assert every young boy does like popcorn.
Assert each of John, Bill, Sam and Dick is a young boy.

Go attack some enemy ship.          (picks one such ship)
Assert some boy does like popcorn.  (picks one boy)
```

A sentence which is used as a condition can also contain pseudo-terms. The **every** and **each of** pseudo-terms will cause the sentence to be tested for each element indicated. The sentence will be true only if each element passes the test. The **some** and **one of** pseudo-terms will cause the sentence to be true if any one of the elements indicated passes the test. The rules below demonstrate these points:

```
If every student is present, go begin class.
If each of John Jones and Dave Jones did get high marks,
        go commend Mr Jones for excellent results.

If some enemy ship did attack some friendly ship,
        go declare war.
If one of John, Bill or Dave did get low marks,
        go discuss parenthood with Mr Jones.
```

Sentences and actions can contain more than one pseudo-term, in which case the scanning operations are nested appropriately. For example, the following two rules are equivalent:

```
Assert every boy does like every girl.

For each boy, for each girl,
   assert that boy does like that girl.
```

4.4. VERB PHRASES

Five basic verb phrases are permitted by ROSIE. Each form captures a specific class of English usage. The verb phrase forms and examples of their use follow.

Class Membership:

(¦ **is**¦**was**¦**will** ¦) [**not**] [**be**] **a**[**n**] relation-name {¦ prep element ¦}

Examples: **is a** doctor
will not be a witness
was an individual **with** glasses

Predication:

(¦ **is**¦**was**¦**will** ¦) [**not**] [**be**] relation-name {¦ prep element ¦}

Examples: **is** happy
was not alone **at the** time
will be late **at** (**the** sound) **of the** bell

Complement:

(¦ **is**¦**was**¦**will** ¦) [**not**] [**be**] relation-name element {¦ prep element ¦}

Examples: **is** nuclear powered
was not really exciting **to** Mary
will be running rapidly **toward** Bethlehem

Intransitive Verbs:

(¦ **did**¦**does**¦**will** ¦) [**not**] relation-name {¦ prep element ¦}

Examples: **did** eat
does eat **with a** fork
will not eat **without a** fuss

Transitive Verbs:

(¦ **did**¦**does**¦**will** ¦) [**not**] relation-name element {¦ prep element ¦}

Examples: **does** study biology **at** school
will cook (**a** steak) **for** dinner
did not divide **by** 2

NOTE: The parentheses in the examples above are needed to disambiguate the scope of the modifying prepositional phrases.

NOTE: Although we refer to **was a** and **will be** under class membership, only **is a** relations are tested for determining class membership. Thus, asserting "Mary **will be a** girl" will not result in "MARY" when testing the database **for each** girl. Similarly, only the present tense **is** forms the predicate is-adjective.

NOTE: Although not shown above, the plural forms of the verbs are permitted as in section 3.2.

4.5. RELATIVE CLAUSE FORMS

The relative clause forms serve two distinct functions. In actions, they permit the assertion of supplementary relations. In conditions, they permit the testing of additional constraints. The relative clause forms and examples of their use follow.

(¦ **where** ¦ **such that** ¦) sentence -- the sentence is tested as a condition which must be satisfied for each element. The sentence can refer to the element tested using description variables or the "**that** relation-name term" form as demonstrated in these rules:

Display every employee (e) **such that**
e **does** play tennis.

Display every employee **such that**
that employee **does** play tennis.

Assert john **is a** man **where** john **is** happy.

NOTE: The assertion above will result in the addition of "JOHN is a man" and "JOHN is happy" to the database. A conditional beginning with "if john

is a man (where john is happy)..." will cause ROSIE to search for just those two assertions. On the other hand, a conditional beginning with "if john is happy..." alone will also result in a successful database search, since the modifying relationship between "John is a man" and "John is happy" is lost.

(¦ **where** ¦ **such that** ¦) (condition) -- the entire condition (which must be enclosed in parentheses) is tested for each element. This rule demonstrates:

Display every employee **where**
(**that** employee **does** play tennis **or**
that employee **does** play volleyball).

Assert john **is a** man **where**
(john **is** happy **and**
john **does** meditate).

(¦ **that** ¦ **who** ¦ **which** ¦) verbphrase -- the verbphrase is treated as a sentence with the element being tested as its first argument. Examples:

Display every employee **who does** play tennis.

Display every distance **which is greater than** 400 miles.

Assert john **is an** employee **who does** play tennis.

(¦ **that** ¦ **who** ¦ **which** ¦) term (¦ **is** ¦ **was** ¦ **will** ¦) [**not**] [**be**] relation-name { preposition element} -- the element tested is passed to the sentence as its second argument as if it were the object of the predicate complement. Examples:

Display every diplomat **such that**
Reagan **is** meeting **that** diplomat **on** Thursday.

Display every diplomat **who** Reagan **is** meeting **on** Thursday.

Assert John **is a** diplomat **who** Reagan **is** meeting.

(¦ **that** ¦ **who** ¦ **which** ¦) term (¦ **did** ¦ **does** ¦ **will** ¦) [**not**] relation-name { preposition term} -- the element tested is passed to the sentence as its second argument, as if it were the verb's direct object. Examples:

Display every girl **where** John **does not** like **that** girl.

Display every girl **who** John **does not** like.

Assert Jane **is a** person **that** Bill **does** know.

preposition (¦ **which** ¦ **whom** ¦) primitive sentence -- this form inserts a prepositional phrase into the primitive sentence. The phrase refers to the element tested and bears the preposition indicated. Examples:

Display every club **such that** John **is a** member **of that** club.

Display every club **of which** John **is a** member.

Assert Boy Scouts **is a** society **of which** John **is a** member.

whose description **is** [**not**] value -- represents the sentence "value **is a** description," where the prepositional phrase "**of** element" has been inserted into the description. For example, these pairs of rules are equivalent:

Display the fleet **such that**
Ship23 **is a** flagship **of that** fleet.
Display the fleet **whose** flagship **is** Ship23.

Display every city
where 2-million **is a** population **of that** city **in** 1981.
Display every city **whose** population **in** 1981 **is** 2-million.

Assert Barrington **is a** town
where BHS **is a** school **of that** town.
Assert Barrington **is a** town **whose** school **is** BHS.

NOTE: The value in a **whose** construct must be a single atom, i.e., numbers, compound names, tuples, etc., are not permitted. The reason for this restriction is the issue of consistency. Since "assert age is 2" is

not allowed in ROSIE, neither is "assert John is a man whose age is 2."

Multiple relative clauses can be logically combined using **and** and **or**. These are examples of rules which use compound relative clauses:

Display every city **which does** support music
and which is not smoggy.

Display every student **of** math **at** Roosevelt High
who did get excellent marks
or who does bring apples **to the** teacher.

Assert john **is a man who is** happy and **who is** rich.

4.6. PRIMITIVE SENTENCES, PROPOSITIONS, AND SENTENCES

The largest syntactical unit that can be added to a database is the primitive sentence. However, ROSIE permits the expression of more complex sentences. In order to add these to a database, ROSIE must first decompose the more complex sentence into a set of primitive sentences. Conditions, actions, rules, and rulesets all incorporate sentences to effect their results. The constructs discussed so far (e.g., terms, descriptions, verb phrases, etc.) are the building blocks of ROSIE sentences--together they combine to provide a rich and powerful expressive ability.

4.6.1. Primitive Sentences

The simplest ROSIE sentences are those that determine a single relationship. These sentences are constructed using the legal relational forms (3.2) with terms in place of elements. Most terms evaluate to single elements, and so these sentences define a relationship determined by the relational form and the element represented by the embedded term.

Primitive sentence forms:

term (¦ **was**¦**were** ¦) [**not**] a[**n**] relation-name {¦ prep term ¦}
term (¦ **is**¦**am**¦**are** ¦) [**not**] a[**n**] relation-name {¦ prep term ¦}
term **will** [**not**] **be** a[**n**] relation-name {¦ prep term ¦}

term (¦ **was**¦**were** ¦) [**not**] relation-name {¦ prep term ¦}
term (¦ **is**¦**am**¦**are** ¦) [**not**] relation-name {¦ prep term ¦}
term **will** [**not**] **be** relation-name {¦ prep term ¦}

term **did** [**not**] relation-name {¦ preposition term ¦}
term (¦ **do**¦**does** ¦) [**not**] relation-name {¦ prep term ¦}
term **will** [**not**] relation-name {¦ prep term ¦}

term (¦ **was**¦**were** ¦) [**not**] relation-name term {¦ prep term ¦}
term (¦ **is**¦**am**¦**are** ¦) [**not**] relation-name term {¦ prep term ¦}
term **will** [**not**] **be** relation-name term {¦ prep term ¦}

term **did** [**not**] relation-name term {¦ prep term ¦}
term (¦ **do**¦**does** ¦) [**not**] relation-name term {¦ prep term ¦}
term **will** [**not**] relation-name term {¦ prep term ¦}

Examples of primitive sentences:

John is a man
The student **did not** fail **the** exam
John **does** support **the** republican candidate
Every boy **does** like **some** girl
John**'s** father **will not** succeed **in** business
Any friendly ship **was** attacked **before** 1300 hours

4.6.2. Propositions

Proposition terms evaluate to proposition elements. They are primitive sentence forms enclosed by an accent grave (ASCII 140) and a single quote. They can represent any legal primitive sentences. When a proposition is evaluated, each term in the constituent primitive sentence is evaluated.

Examples:
`John Smith **was** late **for** work'
`**The** teacher **did** punish **the** student **in** class'
`3 + 4 **is a** prime'

The evaluation name of a proposition element is the relationship represented between "`" and "'". The evaluation name contains the evaluation name for all elements involved in the relationship connected by words which describe the relational form. Thus, the evaluation name for the previous examples might be

`JOHN SMITH **was** late **for** WORK'
`MARTHA did punish JOHN **in** CLASS'
`7 **is a** prime'

Two propositions are equal or equivalent if both propositions represent the same relationship over equal elements.

4.6.3. Comparative Sentences: <, >, = etc.

In addition to the primitive sentences, a few built-in sentence forms allow comparisons of numbers and other elements. Element equality is tested using the equality sentence form, which can be written tersely using the "=" character or in expanded natural English as **is equal to**.

Equality sentence forms:

term **is** [**not**] **equal to** term
term [~]= term

When number elements are tested for equality, they are equal only if both have the same units or labels and represent the same numeric value. Numbers which have different units or labels are not equal regardless of their numeric values. For example, these sentences will all test true:

```
33.2 = 33.2
33 miles/hour = 33 miles/hour
33 miles = 33.0 miles
Probability .4 = Probability .40

33 apples ~= 33
33 apples ~= 33 oranges
33 apples ~= certainty 33
```

The following sentence forms are used to compare number elements only. They also have long and short forms. Comparisons can be made only between numbers with the same units or labels. All other comparisons are illegal and will generate errors.

Number comparison sentence forms:

term **is** [**not**] **greater than** [**or equal to**] term
term [~] > [=] term

term **is** [**not**] **less than** [**or equal to**] term
term [~] < [=] term

NOTE: Comparative sentences are not primitive sentences and therefore they cannot be used in propositions or asserted into the database.

4.6.4. Other Sentence Forms

The remaining legal sentence forms make a variety of tests which may or may not involve the database. Some are supplied for convenience, others expand ROSIE's capabilities in fundamental ways.

NOTE: Like comparative sentences, these forms are not primitive sentences and therefore cannot be used in propositions or asserted into the database.

there is (¦ **no** ¦ **a[n** ¦ **just one** ¦ **more than one** ¦) description -- This is a convenient way to test the cardinality of a class of elements. The description generates as many elements as the test requires. The **no** and **a[n]** options try to generate one element. If an element exists, the **no** test fails and the **a[n]** test succeeds. The **just one** and **more than one** options try to generate two elements. The **just one** alternative fails if none or more than one element exists. The **more than one** alternative succeeds if at least two elements exist. The following rules demonstrate the use of this sentential form:

If there is a file for **the** employee, **go** print **that** file.

If there is no file for **the** employee, **go** request data.

If there is just one enemy ship, **go** attack **that** ship.

If there is more than one enemy ship, **go** surrender.

term **has** (¦ **no** ¦ **a[n** ¦ **just one** ¦ **more than one** ¦) description -- This sentence also performs a cardinality test, but adds on "**of** term" as a prepositional phrase to the description. For example, the following sentence pairs are equivalent:

John **has a** girlfriend
There is a girlfriend **of** John

Grotz Airfield **has more than one** runway **in** use
There is more than one runway **of** Grotz Airfield **in** use

term **is** [**not**] **a[n]** description -- This sentence tests the element generated by the term against the constraints of the description. All

relationships must test true for the test to succeed. The **not** option negates the result of the test. If the tested sentence is a primitive sentence, it tests true only when a sentence in the database matches it exactly or when a predicate matching the sentence concludes true.

If John **is an** exemplary student **of** math, **display** John.

If John **is not an** exemplary student **who did** pass **the** final **and who did** pass **the** midterm, **go** disqualify John.

term **is** [**not**] **provably** (¦ **true** ¦ **false** ¦) -- This sentence allows proposition elements to be tested against the database. A proposition is provably true if it is found in the database, and it is provably false if its negation is found. If neither the positive nor the negative of the proposition exists in the database, the test **is not provably true** and the test **is not provably false** will both test true. In short, ROSIE incorporates a three-valued logic. Like all database access operations, this test will invoke a predicate ruleset if one is defined to test the relationship. When a predicate is invoked, the sentence is considered provably true if the predicate concludes true and provably false if it concludes false. If the predicate reaches no conclusion, the relationship is neither provably true nor provably false (undecided). The following groups of sentences are equivalent:

John **is a** student.
`John **is a** student' **is provably true.**
`John **is not a** student' **is provably false.**

John **is not** exemplary.
`John **is not** exemplary' **is provably true.**
`John **is** exemplary' **is provably false.**

term **is** [**not**] **matched by** pattern -- This sentence uses the pattern matcher to match a string against a pattern. If the element value of term is not a string, it is converted to one using the element's evaluation name. The following rule demonstrates:

If "Dear John;" **is matched by**
{"Dear ", **anything** (**bind** s), ";"},
display NAME **and display** s.

4.7. CONDITIONAL SENTENCES

Conditions are constructs that ask questions. They represent conditions that can be confirmed or refuted by a search for the relevant relationships in the database. A condition is true if all of the constraints on the database test true; otherwise it is false.

Conditions are fundamental in ROSIE because rules are often stated as conditional operations that perform actions only when certain conditions hold. These are examples of rules that test conditions before executing their actions:

If John **did** fail **in** math, **go** expel John.

If any student **did** fail **the** exam, **display that** student.

If the student **was** tardy **or the** student **did** fail **any** exam,
go punish **that** student **and display that** student**'s** name.

If there is an enemy ship **which is** currently **within** range,
go attack **that** ship.

4.7.1. Conditions and Compound Conditions

Conditions are composed of one or more sentences, each of which tests a relationship or invokes a predicate. The conclusion reached by the invoked predicate is used as the result of that test. Sentences are combined using **and** and **or** to create composite logical predications. (The conjunction **and** binds more strongly than **or**.) In addition, conditions can include commas and parentheses to indicate logical groupings. The following examples are compound conditions:

John **does** like Mary **and** Mary **does** like John
John **did** fail **any** exam **or** John **did** fail **any** midterm

The precedence rules permit more complex conditionals to be written without the need for parentheses. Thus,

if John **does** like Mary
and Mary **does** like John
or Mary **does** like Tony

is equivalent to

```
if (John does like Mary
        and Mary does like John)
        or Mary does like Tony
```

Parentheses can logically group sentences to override the built-in precedence groups. In the example below, "John is a student" is grouped with "John is a teacher." Normal precedence rules would have grouped "John is a teacher" with "John did attend graduation."

```
(John is a student or John is a teacher) and
        John did attend graduation
```

Commas can also be used to structure conditions. They are less general than parentheses but will suffice for most applications. Conditions structured with commas are more English-like than those which use parentheses and so should be used whenever possible.

A comma directly following a sentence both ends and begins a sequence of sentences which are grouped as though enclosed in parentheses. The sequence ends with the next comma or with the end of that condition. For example, these groups of conditions are equivalent:

```
(John is a student or John is a teacher) and
        (John did attend graduation)
John is a student or John is a teacher, and
        John did attend graduation

(John is an exemplary student) or
        (John is an average student
        and John's grades did improve)
John is an exemplary student, or
        John is an average student
        and John's grades did improve

(x is an integer or x is a floating-point) and
        (x is positive or x is negative) and
        (x is odd or x is even)
x is an integer or x is a floating-point, and
        x is positive or x is negative, and
        x is odd or x is even
```

NOTE: The syntax of the **if** rule requires a single action. If the execution of more than one action is dependent upon the "if," the set of

actions conjoined with **and** must be either enclosed in parentheses or preceded by a comma (i.e. "if <conditional>, <action1> and <action2>." or "if <conditional> (<action1> and <action2>).").

4.8. ACTIONS

Actions do the work in ROSIE programs. There are many built-in actions. Actions can change the database, build program files, read and write to files and other devices, test conditions, and perform other operations required by a program.

Procedure rulesets (5.3) are invoked by the **go** and **call** actions. A procedure is essentially a user-defined action, and so the number of actions can be expanded with the definition of custom rulesets for special applications.

Action syntax forms are designed to read like English and to indicate the type of operation they perform. Procedure invocation can be equally readable when procedure names and parameters are selected carefully. Multiple actions can be sequenced using **and**. The readability of sequenced actions can be enhanced through the use of parentheses and commas.

A few action forms include other actions as parameters. For example, **if** condition action [**otherwise** action] takes one or two actions as arguments. The ROSIE syntax allows actions to be grouped together with parentheses, commas, semicolons, and **and** to improve readability and to allow the parser to parse rules unambiguously (see 4.8.1).

ROSIE-defined actions occur at all levels of the ROSIE environment. A discussion of each class of actions can be found in the following sections:

Database Actions 3.7

Program Control Actions 5.13

Input/Output Actions 5.12

4.8.1. Actionblocks, Commablocks, Colonblocks

Actionblocks are simply a way of collecting actions together. Commablocks and colonblocks are special types of actionblocks. The commablock can be used instead of parentheses to disambiguate the meaning of a sentence. The colonblock is used in certain types of program control actions.

An actionblock is simply a sequence of one or more actions connected with the word **and**. The following are examples of actionblocks.

```
Display HELP
Display HELP and quit
Display HELP and display the error and quit
```

For example, a rule consists of actionblock followed by a terminal character. These are legal rules:

```
Display HELP!
Display HELP and quit.
Display HELP and display the error and quit.
```

Parentheses can be used to specify several actions at once wherever a single action is required. Any actionblock surrounded by parentheses is a legal action.

Sample actions:

```
(display 1 and display 2)
(assert John is a teacher and deny John is a student)
(send {"Computing...", return} and go compute)
```

A commablock is an actionblock which is preceded with a comma and ended with a comma or period. Since commas are more English-like than parentheses, these constructs were conceived to allow users to avoid parentheses in most applications. The following pairs of equivalent rules demonstrate how commas can be used where parentheses would normally be required. In all of these examples, the absence of grouping characters would change the meaning of the rule.

```
If John is good
  (go commend John and go notify John's parents).
If John is good,
  go commend John and go notify John's parents.

(For each student
  go review that student) and display FINISHED.
For each student,
  go review that student, and display FINISHED.
```

```
For each student
       (if that student did finish
              (go collect that student's paper and
               assert that student's paper was collected)
        and go assign homework to that student).
For each student,
       if that student did finish,
              go collect that student's paper and
              assert that student's paper was collected,
       and go assign homework to that student.
```

Commas often improve readability even when not strictly required, as in the following examples:

```
If the ship was attacked before 1200 hours, go declare war.

For each student, if that student did fail, go expel that student.
```

NOTE: The use of commas does affect the meaning of a sentence. In "If John was late, go scold John and assert John was punished.", a comma is necessary to execute both actions only if the condition is true. The effect of including a comma is illustrated by these equivalent pairs of rules:

```
If John was late
   go scold John and assert John was punished.
(If John was late
   go scold John) and (assert John was punished).

If John was late,
    go scold John and assert John was punished.
If John was late
     (go scold John and assert John was punished).
```

Colonblocks are actionblocks which are optionally terminated by a semicolon. They are used in a few actions which take many action arguments and are therefore clarified by the use of semicolons. These are examples of rules which use colonblocks:

```
Select situation:
        If the target was enemy,    go declare war;
        If the target was neutral,  go apologize;
        If the target was friendly, go hide.

Match the message:
        {"Lovingly", anything}   let the mood be intimate;
        {"Sincerely", anything}  let the mood be concerned;
        {"Cordially", anything}  let the mood be formal;
        Default: let the mood be neutral.
```

4.8.2. Sentences and Modification

Assert and **deny** are actions which take sentences as arguments. Each sentence form instructs **assert** or **deny** to make a particular change or changes to the database. Not all sentence forms can be asserted or denied; those that are legal are documented below.

Primitive sentences -- the relationship indicated is simply added or removed from the database. When a relationship is added, its negation is automatically removed to maintain consistency (see 3.4).

```
Assert John is a student and John does like Mary.

Assert Ship 34 did attack Ship 45 at 400 hours.
```

term **is** [**not**] **a**[**n**] description -- in the absence of the **not** option, all relationships indicated by the description are added or removed at once. When not is used, the roles of **assert** and **deny** are reversed for consistency. The following groups of rules are equivalent:

```
Assert John is good and John is a boy.
Assert John is a good boy.
Deny John is not a good boy.

Assert John is a boy and John does like girls.
Assert John is a boy who does like girls.
Deny John is not a boy who does like girls.

Deny John is good and John is a student.
Deny John is a good student.
Assert John is not a good student.
```

term **is** [**not**] **provably** (**true** ¦ **false**) -- term must evaluate to a proposition element. Without the **not** option, the proposition is simply asserted or denied if provably true is used. If provably false is used instead, the negation of that proposition is asserted or denied. When **not** also appears, the roles of assert and deny are reversed for consistency. The following groups of rules are equivalent:

Assert John **is** good.
Assert `John **is** good' **is provably true.**
Assert `John **is not** good' **is provably false.**

Assert John **is not** good.
Assert `John **is not** good' **is provably true.**
Assert `John **is good**' **is provably false.**

Deny John **is** good.
Deny `John **is** good' **is provably true.**
Assert `John **is** good' **is not provably true.**

NOTE: All other sentence forms are illegal in assert and deny statements and will generate error messages.

4.9. PATTERNS

Patterns are constructs which allow programs to create and manipulate strings of text. A pattern is a sequence of subpatterns enclosed in braces "{}" and separated by commas. Each subpattern in turn represents a restriction on the successive portions of the text string. That text string is either being created from the pattern or matched against the pattern. For example, the subpattern "3 blanks" represents a sequence of three blank characters, and the subpattern "one or more numbers" represents a sequence of one or more numeric digits.

When a pattern is used to create a string, each subpattern is interpreted as a substring of text. These substrings are concatenated together to form the resulting string. For example, each of these patterns can be used to create the string which follows it:

{"The value is: ", 3 + 7}	"The value is: 10"
{John Doe, 3 **blanks**, "Accounting"}	"JOHN DOE Accounting"
{**the** student, " was absent"}	"JANE was absent"

When a pattern is matched against an existing string of text, each subpattern represents a restriction on a portion of the string being matched. Variables can optionally be bound to any substring matched by a subpattern. This allows programs to extract fields of text from strings, using the pattern matching facilities. For example, each of these patterns will match the string which follows it:

```
{"Dear ", anything (bind name), ";"}        "Dear John;"
{2 letters, 3 blanks, 2 numbers}            "ab   34"
{3 numbers, "-", 2 numbers, "-", 4 numbers} "566-96-9990"
```

4.9.1. Subpatterns

This section describes each of the legal subpatterns.

term -- the term is evaluated and the element generated is converted to a string. For matching, that string must appear in the text being matched. For string creation, the string text is inserted into new string.

integer [**or** (¦ **more** ¦ **less** ¦)] **line[s]** -- matches the indicated number of text lines. A line of text is any number of characters followed by a carriage-return or end-of-line character. For string creation, inserts the indicated number of carriage-returns into new string.

integer [**or** (¦ **more** ¦ **less** ¦)] **restriction** [[**not**] **in string**] -- matches the indicated number of characters which satisfy the restriction (see 4.9.2). If the **in string** option is included, the characters matched must also appear in that string. If the **not in string** option is used, the characters matched must not appear in that string. For string creation, this causes the insertion of the indicated number of characters into the generated string (the **not in string** option is disallowed for this case).

anything -- same as 0 or more characters.

something -- same as 1 or more characters.

quote -- matches the double-quote character. For string creation, inserts a double-quote into the new string.

control string -- characters in string are converted to control characters. For example, **send** {**control** "G"} will beep the user's terminal.

codes ((¦ integer {¦ , integer ¦} ¦)) -- represents the characters indicated by the integers interpreted as ASCII character codes.

one of (¦ subpattern {¦ , subpattern ¦} ¦) -- matches any substring that is matched by any one of the subpatterns given.

each of (¦ subpattern {¦ , subpattern ¦} ¦) -- matches any substring that is matched by each of the subpatterns given.

pattern -- invokes the pattern matcher recursively. This subpattern matches any substring which is matched by the pattern given.

return -- matches a carriage-return or end-of-line character. For string creation, inserts a carriage-return or end-of-line into the new string. In TOPS-20, ports and files each use a different end-of-line character, and so **return** will work differently in the **send** action depending on whether the device is a port, a file, or a terminal.

end -- matches end-of-file from a file, port, or terminal. Useful for reading from a device until end-of-file is detected. The end-of-file concept is explained under documentation for the **read** action. End-of-file is actually represented as the NUL character (octal 0), and so this subpattern is equivalent to **codes** (0). No other subpattern will match the NUL character.

4.9.2. Character Restrictions

These are the legal character restrictions. For restrictions which are optionally preceded by **non**, the meaning is inverted.

[non]letter [s] -- any alphabetic characters.

[non]number[s] -- any numeric digits.

[non]alphanumeric[s] -- allows numbers or letters.

[non]blank[s] -- blank characters only.

[non]control[s] -- control characters only.

character[s] -- allows any character.

4.9.3. Variable Binding

A subpattern can be followed optionally by a bind form. This causes the variable specified to be bound to the substring matched by that subpattern. The syntax of **bind** is

(¦ **bind** var [**to the** (¦ **number** ¦ **string** ¦ **name** ¦)] ¦)

The **to the** (¦ **number** ¦ **string** ¦ **names** ¦) option specifies an element-type conversion. The **number** option will try to interpret the substring as a number element, and the variable will be bound to the resulting number. The **name** option will convert the substring to a name element if possible. The **string** option will bind the variable to a string element; if no element type is specified, **to the string** is the default.

4.9.4. Pattern Matching

When a pattern is used to match a string, the entire string must be accounted for by the pattern. Each subpattern will match the shortest segment of the string. For example, consider the following rule:

```
If "[LINE 1] [LINE 2] [LINE 3] " is matched by
        {"[", anything (bind string), "]", anything},
        display string.
```

When this rule is executed, it will display the string "LINE 1," not "LINE 1] [LINE 2] [LINE 3". In other words, the first subpattern was associated with the shortest matching substring.

5. PROGRAMMING STRUCTURES

5.1. OVERVIEW

Like other high-level programming languages, ROSIE allows the user to gather individual ROSIE rules into rulesets. The sorts of constructs discussed so far (e.g., conditionals, assertions, etc.) comprise only a small fraction of the possible actions that ROSIE can perform. Rules in general are discussed in more detail in section 5.2. Other types of actions needed in almost any programming application include input/output (5.12) and control structures such as loops (5.13). Rules, input/output actions, and control structuring actions are the building blocks of rulesets (5.3). A ROSIE program is really a collection of rulesets that examine and modify the information in the database.

5.2. RULES AND RULE VARIABLES

Rules are the fundamental building blocks of ROSIE. Each rule represents one or more actions to be performed when the rule is executed. Users, when interacting with ROSIE, type individual rules to the top level. These are immediately parsed (see 6.1) and executed. On the other hand, ruleset rules are executed only when the ruleset is invoked. Single rules (i.e., not in rulesets) that occur in a previously parsed file are executed when that file is loaded.

A rule consists of one or more actions connected by **and** and terminated by "." or "!". There are also a handful of special interactive rules which end with the terminal character "?" and which are primarily intended for use at the top level. However, they may be used in a ruleset (see 7.4).

These are examples of ROSIE rules:

```
If John is a Republican, go convert John.
Display the object and assert that object was displayed.
For each former president who did serve any term which
        is greater than 4 years, display that president.
Which man was a Republican in office?
Go buy groceries!
```

Variables exist only within the context of a rule. They serve as temporary storage for intermediate results. Variables can be used explicitly or implicitly through anaphora. In either case, a variable's

value always consists of a single element. The value that is bound to the variable results from some operation such as database lookup, reading a string from a file, or finding a substring within a string using the pattern-matching facilities.

When used explicitly, variables must first appear in some variable-binding construct. There are currently only two such constructs: descriptions and patterns. Once a word appears in a bind construct, ROSIE will interpret that word as a variable anywhere else in the rule.

Sample session:

```
            [ ROSIE  Sunday, February 1, 1981  6:54pm ]

<1> If "Dear John," is matched by {"Dear ", anything
         (bind str), ","},
            display Letter_to and display str.
LETTER_TO
"John"
<2> Assert John is a man.
<3> If there is a man (m1), display MAN and display m1.
MAN
JOHN
<4> If there is a man, display that man.
JOHN
<5> logout.
```

In line <1>, the variable STR appears within a bind construct, so STR is treated as a variable rather than as a name. Note that LETTER_TO is treated as a name. In line <3>, M1 appears in a description as a variable, so it is treated as a variable later on.

Line <4> is an example of the use of implicit variables. The phrase "that MAN" refers to the implicit variable bound by the description "MAN." Only descriptions have implicit variables associated with them, and these variables are bound according to rules outlined in section 4.2.4.

NOTE: A variable is bound, implicitly or explicitly, only within a single rule. In the example above, reference to m1 in <4> would have treated M1 simply as a name, rather than as a variable bound to some man.

NOTE: When a relation is restricted by a prepositional phrase (e.g., "man on Mars"), the variable to be bound occurs before the preposition, e.g.,

Display every man (m) on Mars and assert m is an astronaut.

This form also holds for other types of restrictions, as in

Display every man (m) who is tall and assert m is handsome.

NOTE: There are instances in which anaphoric reference may be permitted syntactically but will result in the run-time error "unbound local variable." Thus, if we say "For each person whose age (n) is not five, display <that person, n>" we are searching the database for only those persons about whom the assertion "age is not five" has been made. Thus, there is no value reflecting the actual age available to us through this construct and the variable, n, cannot be bound.

5.3. RULESETS

Rulesets are collections of rules which embody various kinds of procedural knowledge. Like subroutines in more conventional programming languages, they provide a convenient way to modularize a program into coherent procedural units. Unlike other languages, these modules are invoked in a natural way using English-like syntactic forms, and consist of a set of readable rules that modularize information.

There are three kinds of rulesets, each providing a different way of using procedural knowledge. Rulesets can simply carry out a set of operations (procedures), generate a sequence of elements on request (generators), or prove or disprove simple relationships among elements (predicates). In addition, system rulesets can be defined. These are written in INTERLISP and may be desirable because of a need for speed or flexibility.

5.4. PROCEDURES

Procedures are rulesets for performing modular tasks. They accept any number of parameters but do not return results to the rule that invoked them. They allow users to define conceptually modular tasks that can be parameterized conveniently.

Procedures are invoked using the **go** and **call** actions. The **go** action directly invokes the specified procedure. The **call** action provides a way of computing the name of the procedure at run-time. A procedure terminates when the **return** action is executed or the **end** statement is reached. Control is then returned to the rule following the invoking **go** or **call** action.

The following examples invoke a procedure named "display". The procedure looks up the file for an employee and displays an item from the file.

```
Go display Social_security_number for John Doe.
Call the display-procedure using Social_security_number
        for John Doe.
```

Sample procedure:

```
To DISPLAY ITEM for EMPLOYEE:

[1] Send {"The ", the item, " for employee ",
          the employee, 2 lines}.

[2] If the employee has a file,
        display that file's slot for the item,
        otherwise send {"No file for employee: ",
                        the employee, return}.

End.
```

5.5. GENERATORS

Generators are rulesets that define a class of elements procedurally and produce elements one-by-one on demand.

The action **produce** is used by generators to return an element. Generators need not be concerned about generating the same element twice, since an internal check automatically ensures that each element produced is unique. Generators terminate upon executing the **return** action or reaching the **end** statement.

Generators are invoked whenever the database is requested to generate the members of a class. As described in 4.2.5, this happens when a description is used as a generator. The following examples all use the simple description "project_leader" as a generator. The first asks for just one leader, the next tries to generate two, and the last will generate and print as many as possible:

```
Display the project_leader.
If there is more than one project_leader, display YES.
Display every project_leader.
```

If the generator below is defined when these rules are executed, it may be invoked.

Sample generator:

```
To generate PROJECT_LEADER.

[1] Produce the current research_director.

[1] For each active project,
        produce the employee who does head that project.

[2] For each project which will begin before 1982,
        produce the employee who will head that project.

End.
```

NOTE: If a generator is defined for a particular description and the database already contains "is a" relations for that class, those relations will be accessed first. Thus, if an element satisfying **is a**(project_leader) exists in the database, the example "Display the project_leader" given above would never cause the invocation of the generator.

5.6. PREDICATES

Predicates are rulesets which determine the truth or falsehood of relationships among elements. They allow the specification of a method for testing relationships that cannot or should not be stored explicitly in the database.

The **conclude** action terminates a predicate by specifying whether the relationship is either provably true or provably false. If, instead, the predicate is terminated with a **return** statement, the relationship is undecided (neither true nor false). The same occurs when the **end** statement is reached.

Predicates may be invoked whenever the database is asked about a relationship whose relational form matches that of a defined predicate. If the test cannot be resolved by a simple database search, the appropriate predicate is invoked and its conclusion determines the result of the test. The rules in the following examples all test relationships of the form "element is active in element":

```
If John Doe is active in GLEE_CLUB, display YES.
Display every employee who is active in some
     organization.
For each organization in which John Doe is active,
        display that organization.
```

The following example defines a predicate that determines whether an employee "is active in an organization." If the above rules are executed after this predicate is defined, the predicate will be called once by the first rule and possibly many times by the second and third.

Sample predicate:

```
To decide EMPLOYEE is ACTIVE in ORGANIZATION:

[1] If the employee has a file
        and the organization is currently in
        (that file's slot for ORGANIZATIONS),
        conclude TRUE.

[2] If there is a membership_roster for the organization
        and the employee is currently on that
             membership_roster,
        conclude TRUE.

[3] Conclude FALSE.

End.
```

NOTE: If a predicate incorporates the same general form as a sentence in the database, the sentence (or, possibly, sentences) in the database will be accessed first.

5.7. SYSTEM-DEFINED RULESETS

The ROSIE environment includes a number of predefined rulesets that perform useful operations. This package is called the ruleset library. A brief explanation of each of the system-defined generators, predicates, and procedures is given below. Most of the predefined rulesets are system rulesets, i.e., they are written in INTERLISP. Section 5.11 explains how to write a system ruleset and Appendix C, "System Support Library," gives the actual code for the rulesets described below.

element **is a thing** -- always true regardless of element type.

element **is a proposition** -- true if element is a proposition, otherwise false.

element **is a tuple** -- true if element is a tuple, otherwise false.

element **is a string** -- true if element is a string, otherwise false.

element **is a name** -- true if element is a name, otherwise false.

element **is a number** -- true if element is a number, otherwise false.

element **is a class** -- true if element is a class, otherwise false.

element **is a filesegment** -- true if element is a filesegment, otherwise false.

proposition **is true in** database -- tests the truth of the proposition in the given database; concludes true or false.

element_type of element -- produces a single name element which indicates type of element. The name element produced is one of PROPOSITION, TUPLE, STRING, NAME, NUMBER, CLASS, FILESEGMENT.

integer from integer **to** integer -- produces each integer in the given range.

integer from integer **to** integer **by** integer -- produces integers from the first to the second, incremented by the third.

number_value of number -- produces numeric value of the number element (units or labels are discarded).

negation of element -- if the element is a number, produces the negation of the number. If the element is a proposition, produces the negation of the proposition. Otherwise an error occurs.

square-root of number -- produces square root of number.

sine of number [**in radians**] -- produces sine of number. Number is assumed to be in degrees unless in radians option is given.

cosine of number [**in radians**] -- produces cosine of number. Number is assumed to be in degrees unless in radians option is given.

tangent of number [**in radians**] -- produces tangent of number. Number is assumed to be in degrees unless in radians option is given.

arcsine of number [**in radians**] -- produces arcsine of number in degrees or radians.

arccosine of number [**in radians**] -- produces arccosine of number in degrees or radians.

arctangent of number [**in radians**] -- produces arctangent of number in degrees or radians.

floor of number -- produces number with fractions truncated.

log of number -- produces logarithm of number.

antilog of number -- produces antilog of number.

random number from number **to** number -- produces a single pseudo-random number within bounds.

member of tuple -- produces every element in tuple, in order.

member of tuple **at** integer -- produces the single element which is at position integer in tuple.

length of tuple -- produces length of tuple.

tail of tuple -- produces a copy of tuple without the first element.

tail of tuple **from** integer -- produces a new tuple which contains the elements of old tuple starting from position integer.

concatenation of tuple **with** tuple -- produces a new tuple which contains elements of both tuples in order.

reverse of tuple -- produces a new tuple which contains the elements of the old tuple in reverse order.

add proposition **to** database -- adds the given proposition to the given database.

remove proposition **from** database -- removes the given proposition from the given database.

show database -- displays the contents of the given database.

5.8. PRIVATE RELATIONS IN RULESETS

Because rulesets often need to save intermediate results, each ruleset invocation is allocated a private database for class membership relations. This database is accessible only by that ruleset invocation, and it is discarded when the ruleset has finished its work. A place for storing private classes is useful for a number of reasons. For example, rulesets using this method of storage do not have to clean up after themselves. Also, an error which occurs during ruleset execution need not leave the global database cluttered with relationships that were meant to be discarded.

Global relations and private relations are accessed using exactly the same ROSIE operations. Among these, the **assert**, **deny**, and **let** actions modify relations, descriptions can be used to generate elements from relations, and conditions test for the presence of relations.

A database access operation examines the private **is a** relations if the relation-name of the relationship has been declared private. A relation-name is private if it (a) is the name of a parameter to a ruleset or (b) has been declared **private** in the ruleset definition.

The following procedures demonstrate how private relation-names affect database access operations. Both procedures display a list of candidates for employee benefits by scanning the global database for part-time and full-time employees.

Sample procedure:

```
To PRINT_CANDIDATES:

[1] For each full-time employee,
        assert that employee is a candidate.

[2] For each part-time employee,
        assert that employee is a candidate.

[3] Display every candidate.

[4] Deny every candidate is a candidate.

End.
```

Sample procedure:

```
To PRINT_CANDIDATES:

Private CANDIDATE.
```

[1] **For each** full-time employee,
assert that employee **is a** candidate.

[2] **For each** part-time employee,
assert that employee **is a** candidate.

[3] **Display every** candidate.

End.

In the first procedure, the relationship "element **is a** candidate" is asserted in the global database for each full- or part-time employee by rules [1] and [2], and then all of those employees are displayed by rule [3]. Before the procedure exits, rule [4] cleans up by removing those candidate assertions from the global database.

The second procedure declares candidate as a private relation-name, so the relationship "element **is a** candidate" asserted by rules [1] and [2] are all directed to the "private database." Rule [3] scans the private relations looking for those relationships and printing the employees. Rules [1] and [2] still scan the global database looking for full-time or part-time employees, since neither full-time, part-time, nor employee is declared as a private relation-name. When the procedure exits, no cleanup is done because the global database has not been modified.

To further illustrate, imagine that the following rules appear in a ruleset which declares employee, deserve, and full-time to be private relation-names. All of these rules when executed will scan or modify private relations only.

Declaration: **Private** EMPLOYEE, DESERVE, FULL-TIME.

Display the employee.

If the employee **is** full-time, **display** YES.

Let the employee **be** John Doe.

Display every employee **who does** deserve salary review.

Display every full-time employee **of** Corporate
Enterprises.

For each full-time employee **in** 1969,
assert that employee **does** deserve benefits.

Assert Mike Meyers **is a** full-time employee of DataTech
who does not deserve benefits.

5.9. PASSING ARGUMENTS TO RULESETS

As mentioned above, a ruleset's parameter names are private relation-names. When a ruleset is invoked, relationships of the form "element **is a** parameter-name" are placed in the ruleset's "private database," one for each parameter. Parameter elements can thus be retrieved using normal database access operations.

Most parameters are retrieved from terms embedded in a prepositional phrase. These parameters are passed to rulesets according to the preposition which precedes them, not according to the order in which they appear. That is, prepositional phrases need not present parameters in the order in which they appear in a ruleset definition.

The following examples illustrate the use of parameters. The first rule invokes the procedure PRINT, passing three elements as parameters: a number, a string, and a name. Using the device TEMPFILE, the procedure prints the string and the number and then exits. The prepositional phrase supplies two of the parameters correctly even though the order of the prepositions has been reversed.

Go print 55 miles/hour **to** TEMPFILE **after** "Speed limit:".

Sample procedure:

To PRINT VALUE **after** STR **to** DEVICE:

[1] **Send to the** device {**return**,**the** str,1 **blank**,**the** value,**return**}.

End.

Note that the VALUE, STR, and DEVICE parameters are retrieved in this case using the term "**the** description"; the parameter names act as simple descriptions. This works because the following become private relations when the procedure is invoked:

55 MILES/HOUR **is a** value.
"Speed limit:" **is a** str.
TEMPFILE **is a** device.

5.10. ORDER OF EXECUTION IN RULESETS

The order of execution of rules can vary from ruleset to ruleset. Sometimes it is desirable that the rules be executed sequentially, cyclically, or even randomly. The **execute** action tells ROSIE which type of execution you want. The three forms are **execute sequentially, execute cyclically**, and **execute randomly**.

The **execute** action occurs in the ruleset definition after the declaration of the private relations (if there are any). If no declaration is supplied, ROSIE assumes sequential execution. Most ruleset examples in this document incorporate sequential execution.

Sequential order executes ruleset rules in a top-to-bottom order. After the last rule has been executed, a **return** action is automatically executed terminating the ruleset as though the **return** was executed explicitly. This order is ideal for most rulesets, since it is the most intuitive and since an explicit **return** statement need not be included.

Cyclic order executes rules sequentially, but rather then ending once the last rule is executed, it begins again with the first rule. The ruleset terminates only on the execution of a **return**, **conclude**, or **quit** action. This order is useful when a sequence of rules must be executed repeatedly.

Random order executes rules in a pseudo-random fashion. After each rule is executed, another rule is selected at random and executed. As with the cyclic execution, the ruleset must be terminated explicitly.

5.11. WRITING SYSTEM RULESETS

System rulesets are rulesets which are defined in the ROSIE implementation language (currently INTERLISP). Writing these rulesets requires some knowledge of the implementation language.

System ruleset bodies are single INTERLISP Spread-NLAMBDA expressions. The NLAMBDA expressions should have one parameter for each argument. Arguments are passed to the NLAMBDA expression in alphabetical order of the prepositions associated with each parameter, not in the order in which parameters appear in the ruleset header.

NLAMBDA expressions for system procedures need not return any special value. They are simply executed when the procedure is invoked.

NLAMBDA expressions for system predicates should return the literal atom <TRUE> or <FALSE> to reach a conclusion. Any other returned value is treated as "undecided."

System generators can produce strings, numbers, names, and tuples. If the NLAMBDA expression returns one of these, that element is treated as the only element produced by the generator. The expressions can also return a list of elements. In that case, the list of elements is interpreted as those produced by the generator. NIL is treated as an empty list (no elements generated).

To return a string from a system generator, you need only return an INTERLISP string element. To return a number, you must make sure the number is without label or units. Names are returned as simple atoms. Tuples can be returned by using (LIST-TO-TUPLE x) where x is the list of elements you want in the tuple.

Comments cannot appear between a system ruleset header and the body which follows it. The parser expects to find exactly one s-expression after each system ruleset header.

System rulesets may be written using CLISP. All system rulesets are DWIMified when necessary. The filepackage action **compile** will compile system rulesets.

The INTERLISP function ABORT[MSG1; MSG2] is available to system rulesets. ABORT is the function called by ROSIE when a runtime error occurs. It prints the location of the error followed by any messages supplied as arguments to ABORT. The user is then returned to the top level gracefully. This function is similar to the INTERLISP ERROR function. MSG1 should be a string; MSG2 is an optional list of items which are printed following MSG1.

The following two system rulesets are examples taken from the system ruleset library:

```
System ruleset to generate INTEGER from INT1 to INT2:
(NLAMBDA (INT1 INT2)
        (IF (NULL (FIXP INT1)) THEN
           (ABORT "Not an integer:" (ELTTOTOKENS INT1)))
        (IF (NULL (FIXP INT2)) THEN
           (ABORT "Not an integer:" (ELTTOTOKENS INT2)))
        (FOR I FROM INT1 TO INT2 COLLECT I))

System ruleset to decide ELT is a STRING:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'STRING)
     THEN '<TRUE>
                ELSE '<FALSE>))
```

5.12. INPUT/OUTPUT

The input/output actions allow programs to read and write directory files, communicate with the user's terminal, run other programs, and connect to remote computer sites. Users can also create transcript files that will record all or part of a ROSIE session.

An input/output device is a source/destination which can be read from or written to. The user's terminal is one such device. Other devices can be directory files (files of text kept in the user's directory) or ports (separate jobs logged in to the host computer). Input from these devices is performed using the **read** action. The **send** action is used to output text to a device.

Before a device can be accessed, it must be initialized or opened using one of the **open** actions. The user's terminal is a device which is always open. When a directory file is opened, it is "noticed" by ROSIE and an internal buffer is allocated for it. The file is not actually created or modified in the user's directory until it is closed with the **close** action. Files can be opened to read, write, or append. The **read** and **append** options both open an existing directory file, while the **write** option will create a new file. The **read** action will work only on files which are opened to read. The **send** action allows files to be opened for writing or appending.

When a port is opened, a new job is logged in on a TOPS-20 pseudo-teletype. This job looks just like another user to the host computer. Text sent to the port is read by the computer as though typed by the "user," and the computer's response can be retrieved by reading from that port. Closing the port will logout the phantom user and kill that job. Since ports act just like users, they allow ROSIE programs to do anything a user can do, which includes running another ROSIE program and also network communication programs.

NOTE: If you start a job from within ROSIE you must make sure to follow every **send** to that job by a **read**. Sending characters to a job results in those characters being echoed in the job's output buffer. If you do not read from that output buffer it will fill up and block the process. Once the job has been stopped the input buffer you are sending to will fill up and, eventually, you will receive an input overflow error. You can solve this problem simply by following each **send to the** job {**anything**,**return**} with a **read from the** job {**anything**, **return**, **the** prompt-character}.

Input/output devices are all assigned names when they are opened. Those names are recognized by the **read**, **send**, and **close** actions. The user's terminal, which is always open, has no name and is used when no device is specified. (The user's terminal may be referred to explicitly using "TTY:".) A device can be named with a single-word name or string element.

The commands for input and output actions can be found at the beginning of section 5.14.

5.13. PROGRAM CONTROL STRUCTURES

These actions provide various ways of telling a program what to do and when to do it. Program control structures include iterative loops, conditionals, a variety of structures resembling a "case" statement, procedure invocation, etc. A complete list of program control actions is given in section 5.14.

5.14. PROGRAMMING ACTIONS: QUICK REFERENCE

This section contains descriptions of each of the input, output and program control actions. For a more general discussion of these topics see sections 5.12 and 5.13.

display term -- prints the evaluation name of the term to the user's terminal followed by a carriage return.

open term **to** (¦ **read** ¦ **write** ¦ **append** ¦) -- opens a directory file. Term must evaluate to a single-word name element or a string. The **read** and **append** options open an existing file, the **write** option opens a new file. A file opened to append will be added to by the **send** action rather than overwritten.

open port to term -- creates a new port (logged in and running an EXEC). Term must evaluate to a single-word name or string element, which will be the name of the new port.

close term -- closes the device. Ports are killed, directory files are closed, and the directory is updated. Term must evaluate to a single-word name or string element.

close everything -- closes all open files and ports.

NOTE: If you open a file for both reading and writing, the **close** action will only close the file with respect to one of those functions. To close the file completely you should use the **close everything** action.

send [**to** term] pattern -- creates a string from the pattern and sends it to the device (a name or string). If no device is specified, sends text to the user's terminal.

read [**for** number **seconds**] [**from** term] pattern -- reads text from a device (from the user's terminal if no device is specified). Read will input one character at a time from the device. The input operation terminates after the string thus accumulated matches the pattern supplied. If at any time the pattern matcher recognizes that the string will never match the pattern, an error occurs. The text read can only be retrieved using variables in the pattern; these are bound when a match occurs. If a time limit is supplied, end-of-file is forced for the terminal and ports after the read has used up the allotted time. When no time limit is specified, end-of-file is declared for ports when the port program is waiting for input, while reading from the terminal will never terminate due to end-of-file. End-of-file for directory files occurs after the last character on the file. Note that end-of-file is treated by the pattern matcher as a character (octal 0) which can be matched with the end subpattern. This allows a **read** action to read deliberately until end-of-file from any device without generating an error.

NOTE: There is a problem if you try to **read** the terminal input buffer when there is nothing there. Saying "read for 5 seconds {anything (bind), return}" when the buffer is empty results in the error "input won't match string." One suggestion for getting around this problem is to use the following:

```
Read for 5 seconds {one of ({anything (bind x),
          return (bind y)},
               {anything (bind x), end (bind y)})
   and if y is matched by {end} go failure
       otherwise go success.
```

dribble to term -- term must evaluate to a single-word name element. This action begins copying everything which appears on the user's terminal to the directory file name. The dribble file need not (should not) be opened or closed. This is a convenient way to save a transcript of all or part of a ROSIE session for later viewing. Note that you may edit during a "dribbled" session, but that part of the session will not be dribbled.

stop dribbling-- stops copying to the dribble file.

if condition action [**otherwise** action] -- the condition is tested and if

true, the first action is executed. If the condition is not true and the second action is given, it is executed instead.

NOTE: Unlike other languages, ROSIE does not use the "if...then" format. In fact, the "then" is not even optional. The user must be aware of this because it is not always the case that using a "then" will cause a syntax error. In "if john is bad then display yes," for example, "then" is considered a perfectly valid term (if this seems counterintuitive, consider "if john is growing old display yes"--remember that ROSIE relies on purely syntactic constraints during parsing).

unless condition action [**otherwise** action] -- action occurs only when condition is not probably true. This is useful for actions conditional upon the absence of database sentences.

[**for each** description] [**while** condition] [**until** condition action] -- this construct is used to perform iterative loops. The action is performed repeatedly until the loop is exhausted. If **for each** description is supplied, the action is performed once for each element generated from the description. If **while** condition is given, the condition is tested before each iteration and the loop terminates if the condition is not true. If **until** condition is given, that condition is tested after each iteration, and the loop is terminated if the condition is true. Any combination of **for each**, **while**, and **until** can be used. These rules demonstrate this action:

For each enemy ship,
go attack **that** ship.

For each enemy ship
while there is an unassigned aircraft,
go deploy **that** aircraft **to that** ship.

Until Norswegia **does** surrender,
go attack Norswegia.

match term **against** pattern -- the element value of term is converted to a string if it is not one already; invokes the pattern matcher to match that string against the pattern. If the match succeeds, any variable bindings indicated in the pattern are performed; otherwise this action does nothing.

select term : {¦ tuple colonblock ¦} [**default** : colonblock] -- this action finds the appropriate action block and executes it. Each tuple term

is evaluated and searched for the element value of the first term. If that element is found, the actions associated with the tuple containing it are executed. If no tuple contains the element, the default actions (if supplied) are executed instead. This rule demonstrates:

```
Select the country:
        <Russia, China>           display BAD GUYS;
        <USA, Canada>             display GOOD GUYS;
        <Any third-world nation> display CANT TELL;
        Default: display MORE INFO and go get info.
```

match term : {¦ pattern colonblock ¦} [**default** : colonblock] -- similar to the **select** action, but selects the actions by attempting to match the element value of term against each pattern until a match is found. This rule demonstrates:

```
Match the message:
        {"Hello"}       display HOWDY;
        {"Goodbye"}     display SOLONG and logout;
        {something}     display WHAT?;
        Default:        display SAY SOMETHING!
```

choose situation : {¦ **if** condition colonblock ¦} [**default** : colonblock] -- similar to the **select** action but selects the actions by testing each condition. This rule demonstrates:

```
Choose situation:
        If the mayor is a republican,
                go donate 120 dollars;
        If the mayor is a democrat,
                go donate 300 dollars;
        Default: go donate 100 dollars.
```

go relation-name [term] {¦ preposition term ¦} -- invokes the procedure relation-name with the value of term (if given) and the value of each prepositional term (if any) as arguments.

call term [(**using** term)] {¦ preposition term ¦} -- similar to the **go** action, but uses the value of the first term, which must be a single-word name element, as the name of the procedure to invoke.

do nothing -- does absolutely nothing. Sometimes useful as a place-filler in actions which take action arguments. These equivalent rules demonstrate:

If John **is** happy, **do nothing, otherwise go** cheer-up John.

Unless John **is** happy, **go** cheer-up John.

return -- this action terminates a ruleset. In procedures, it simply ends the ruleset. In generators, it causes the ruleset to stop generating elements. In predicates, it declares the truth value of the relation to be undecided.

produce term -- in generator rulesets, causes the ruleset to return the element value of term.

conclude (**true** ¦ **false**) -- in predicate rulesets, causes the rule set to reach a conclusion of provably true or provably false.

quit [**because** pattern] -- this is a programmable control-B (interrupt). ROSIE immediately returns to the top level. If because pattern is given, the pattern is converted to a string and that string is printed to the terminal before quitting.

wait for number **seconds** -- the program does nothing for the indicated number of seconds. This causes programs to pause for any length of time without using computation resources during that time. Useful when a program must wait for a response from a user or from another program.

save as term -- term must evaluate to a single-word name element. This action creates a file name.EXE which freezes the complete state of the program which executed it. The user can type "name" to the operating system to resume program operation from after the point at which the **save** action was executed. The **save** action closes all input/output devices and creates the name.EXE file, but otherwise does not affect the running program which executed it. For example, ROSIE is created by executing

Sysload SYSTEM **and save as** ROSIE.

revert to term -- term must evaluate to a single-word name element. This action is the inverse of the **save** action. The program which executes it is completely abandoned and the name.EXE file is resumed as though the user typed name directly to the operating system. Provides a programmable way to switch the user to another ROSIE program.

logout -- closes all open input/output devices and terminates the ROSIE session. Control is returned to the operating system or invoking program.

push -- connects user to the operating system (a lower EXEC), where he can interact without losing the current ROSIE session. Return to ROSIE by typing POP to the EXEC.

6. STORING PROGRAMS: THE FILEPACKAGE

6.1. OVERVIEW

The filepackage actions are the heart of the ROSIE programming environment. They help the user build, modify, examine, and keep track of programs in a way that exploits the modular and English-like nature of ROSIE rulesets. They also encourage interactive and real-time system development by minimizing parsing and compiling overhead caused by changes to individual rules.

The filepackage actions also support debugging by helping users to find, display, and modify offending rules. Other helpful features keep track of syntax errors, outline the contents of files, remember the compilation status of rulesets, and aid in locating strings of text in programs.

User programs are stored as program files in the user's directory. Rule numbers in program text are maintained automatically, and text is spaced evenly for easy reading. Along with text files, the filepackage actions maintain other directory files that reduce parsing, compilation, and storage overhead. Thus a single program file is actually a group of files a user modifies via filepackage actions rather than directly.

6.2. PROGRAM FILES

ROSIE programs are kept on program files. These contain ruleset definitions and file rules. Program files are named by the user when they are created by the **build** action. They can be referenced by that name when loaded, compiled, examined, and edited.

To the user, a program file is simply a text file that may be modified. To work on a file, the user loads it using the **load** action. When a file is loaded, its rulesets are defined and its file rules are executed. It is also "noticed" by the filepackage, which means that filepackage actions will know where to find things in that file. The **sysload** action can also be used to load a file without "noticing" it. This is more efficient but does not allow the user to change or examine the file. The system ruleset library is preloaded using **sysload**.

For every program file, the filepackage actually maintains three or four files to minimize overhead. None of these files should ever be edited or deleted directly by the user, since they are all required by the filepackage and are interrelated in unobvious ways. These files are explained briefly below.

Files kept by the filepackage:

.TEXT file -- contains the actual program listing in its original text form. It is the only readable file in the group and is the file which should be printed when a hardcopy program listing is required. The .TEXT file is used by the parser in order to produce the .PARSE file.

.PARSE file -- contains the executable parse of all items in the .TEXT file. When a program file is loaded, this is the file which is actually loaded into the system.

.MAP file -- contains a map of the .TEXT, .PARSE, and .COMPILE files. It allows filepackage actions to access and change portions of the other files efficiently.

.COMPILE file -- is created only when the program file is compiled. It contains compiled code and is loaded in place of the .PARSE file whenever it exists, and its creation date is newer than that of the .PARSE file.

NOTE: A file must be "noticed" in order to be examined or changed. For ROSIE to "notice" a file, it must be loaded or built. A **load** action works on the parse file and will "notice" only those rules and rulesets that are free of syntax errors in the ruleset header. To edit those rules and rulesets that have header errors, you must edit the file (i.e., "edit file <filename>" or "edit <filename>"). Rulesets with syntax errors in the ruleset body are "noticed." The **build** action is used to create a new file; once the file has been edited, and the editing session completed, the file will automatically be parsed and will thereafter behave like a loaded file.

6.3. RULESET DEFINITIONS AND FILE RULES

A program file usually contains ruleset definitions. These rulesets are defined when the ruleset definitions are loaded (using the **load** or **sysload** action). Rulesets which contain syntax errors (other than those in the header) are not defined but are still "noticed" by the filepackage so they can be corrected. Rulesets with errors in the header (e.g., "To generate is a foo" rather than "To decide is a foo") are neither loaded nor noticed; to correct such errors the entire file must be edited.

Program files can also contain file rules, i.e., individual rules which are not part of any ruleset. When a file is loaded, the rules are executed after all of the file's rulesets have been defined. File rules can be used

to initialize the database, start a program, print useful information, etc.

The following example of a program file called INTEGERS will print the integers from 1 to 10 when loaded. It contains three file rules and one ruleset definition. Following the file listing is a sample session which shows the file being loaded and examined.

```
==========
==========

[: INTEGERS  Created 28-Jan-81 1:07pm, edit by GORLIN :]

[rule 1] Let the first number be 1.

[rule 2] Let the last number be 10.

        [ The next rule invokes PRINT_NUMBERS when this file is
            loaded. ]

[rule 3] Go Print_numbers.

        [ This procedure prints a sequence of numbers. ]

To PRINT_NUMBERS:

[1] For each integer from (the first number) to (the last
       number),
          display that integer.

End.

==========
==========
```

Sample session:

```
            [ ROSIE  Friday, January 30, 1981  8:50pm ]

<1> Load INTEGERS.
To PRINT_NUMBERS
1
2
3
4
5
6
```

```
7
8
9
10
<2> Show INTEGERS, 3.

        [ The next invokes PRINT_NUMBERS when this file is
             loaded. ]

[rule 3] Go Print_numbers.

<3> Show PRINT_NUMBERS.

        [ This procedure prints a sequence of numbers. ]

To PRINT_NUMBERS:

[1] For each integer from (the first number) to (the last
     number),
        display that integer.

End.

<4> Logout.
```

In the example above, note the rule numbers printed as comments at the beginning of every file rule and ruleset rule. These comments are inserted and updated automatically by the filepackage and are displayed along with the rules when the user examines the file text. Rule numbers also aid in debugging, since ROSIE error messages identify offending rules by number. The filepackage allows rules to be cited by those numbers using the form **edit** filename rulenumber (see also statement <2> in the sample session).

Rules and other file items are also separated uniformly by blank lines, regardless of how the user types them. This simplifies rule editing and further improves readability.

A comment which appears between rules or rulesets is associated with the item which follows it and is considered part of that item. When a rule or ruleset is displayed or edited, the comments associated with it, if any, are included as part of the item. The example above has two comments; they are parts of file rule 3 and the PRINT_NUMBERS ruleset, respectively. Comments can also appear anywhere within a rule.

6.4. MODIFYING AND USING PROGRAM FILES

Some filepackage actions (**insert**, **edit**, **copy**, etc.) are used to edit program text. These actions all create new .TEXT, .PARSE, and .MAP files reflecting the changes made. Parts of these files that were not changed are copied from the old versions to save time. The parser is called only on portions of program text edited by the user, so the entire file is not reparsed after each modification. Users should take advantage of the fact that editing actions will work on individual rules and rulesets as well as on whole files. Whenever an entire file is edited, that file must be completely reparsed regardless of what changes were made.

NOTE: If you select an editor which associates line numbers with each line (e.g., SOS or DEC's EDIT), you must leave each edit session with a command that does not store the line numbers in the file written by the editor (U for SOS, EU for EDIT). If line numbers are left in, an INTERLISP error will occur.

6.5. COMPILATION

The **compile** action allows users to compile an error-free program file. A compiled file, once loaded, takes up much less storage, and its rulesets will execute much faster. Compilation is therefore critical when a program grows large, since it may not fit into memory in any other form. In fact, it may be necessary to **sysload** compiled forms to reduce the program to an acceptable size.

The **compile** action has two effects: redefinition of all rulesets in the program file with compiled definitions, and creation of a new .COMPILE file. The next time the program file is loaded, that .COMPILE file will be read instead of the .PARSE file, so the compiled definitions of rulesets will be loaded. When compiled program files are changed and recompiled, the **compile** action will recompile only rulesets which have changed since the last compilation and will copy old compiled definitions from the old .COMPILE file to the new one. For this reason, old .COMPILE files should not be deleted.

6.6. FILESEGMENTS

Filesegment terms evaluate to filesegment elements. A filesegment term identifies either a file, a ruleset, or a sequence of file or ruleset rules.

Examples: "**To generate** CONTENTS **of** BOX **at** TIME"
"**To generate** MEMBER **of** TUPLE"
"**To** PRINT FILE, 5"
"**To decide** MAN **is a** COLONEL, 1 7"

NOTE: The system ruleset "Member of tuple" is referred to above like any other ruleset, i.e., you do not need to include the words "System ruleset" from the header when accessing a system ruleset as a filesegment.

The examples above demonstrate the long form of this term. There is also a more convenient short form which allows files or rulesets to be identified with a single word. If the word names a loaded program file, that file is taken to be the source of the segment. Otherwise, the word is assumed to be all or part of a relation-name which identifies a loaded ruleset. If more than one ruleset fits the description, the user is asked which one was intended. For example, the following pairs of terms might be equivalent if the indicated files or rulesets are loaded:

Examples: "To generate CONTENTS of BOX at TIME, 3 9"
CONT, 3 9

"To generate PRINT FILE"
PR

file PROGRAM, 6
PROG, 6

6.7. FILE COMMANDS: QUICK REFERENCE

File and directory actions allow manipulation of directory files from within ROSIE. Their operation is system-dependent, since they attempt to duplicate the basic operating system commands. All of these actions accept single-word names or string elements.

NOTE: These actions require the full file name specification, i.e., "filename.ext." (The **dir** action will work without the extension by assuming all files with the given filename are intended.)

dir [term] -- displays selected file names in the directory, with their creation date and size. Accepts TOPS-20 special characters. Works much like the TOPS-20 DIRECTORY command.

type term -- types contents of file to the terminal.

copy term **to** term -- copies a file.

append term **to** term -- appends contents of file1 to the end of file2. File1 is not changed.

rename term **to** term -- renames a file.

delete term -- deletes a file from the user's directory. Does not ask for confirmation. The file will not actually be removed from the filesystem until the directory is EXPUNGED in TOPS-20.

Filepackage program file actions create, load, and compile program files. As discussed in section 6.2, a program file consists of three or four directory files maintained by the filepackage. These files should only be manipulated using the filepackage actions.

load term -- term can evaluate to a single-word name or string element. Loads a program file into core. Once loaded, rulesets in the file are defined in core and can then be invoked. If a .COMPILE file exists, it is loaded; otherwise the .PARSE file is loaded. If the program file contains file rules, these are executed after rulesets are defined. **Load** also "notices" the file, which allows other filepackage actions to access it. A ruleset that contains a syntax error will still be "noticed" (unless the error was in the ruleheader) but will not be defined.

sysload term -- same as **load** but the program file is not "noticed" and so cannot be displayed or edited. More efficient than **load** and takes up less memory space. **Sysload** is useful when a program file is loaded only for its ruleset definitions or file rules. For example, the system ruleset library is sysloaded.

compile term -- requires single-word name element. Compiles or recompiles a program file, creating a new COMPILE file. If the file has not been loaded, it is loaded first. If the file was previously compiled, only the rulesets which have changed are recompiled, and old compiled definitions are copied from the last .COMPILE file. The **compile** action also redefines each ruleset compiled with the new compiled definition. This frees the space previously used by uncompiled rulesets (compiled rulesets take up very little space). A program file must be free of syntax errors before it can be compiled. File rules are also compiled if any are found, and will subsequently execute much faster when loaded.

build term -- this action creates new program files. Requires a single-word name element. The file will bear the name given to **build**. **Build**

leaves the file loaded and noticed, so the user can immediately begin editing it.

change term **to** term -- this is the only safe way to rename a program file. Both arguments must be single-word name elements. The program file named by the first argument is renamed. The file need not be loaded, but the .MAP file must exist.

parse term -- term must evaluate to a single-word name element. This action can be used to turn a solitary text file into a loadable program file. The text file is parsed and a new set of filepackage files (.TEXT, .PARSE and .MAP) is created for it. The new program file can then be loaded using the **load** or **sysload** action. The **parse** action is used only for special applications where a text file which was not created through the filepackage must be loaded. Also, if a .PARSE or .MAP file is accidentally deleted, the program file can be reconstructed by executing

Parse filename.text.

Most of the filepackage filesegment actions change program files. Each change causes a new .TEXT, .PARSE, and .MAP file to be created in the user's directory. Those actions which invoke the user's text editor will parse the text edited immediately after the editor is exited. The program file is then rewritten, and if no syntax errors have been introduced, any rulesets that have been changed are redefined. The rulenumber comments that appear before each rule are inserted and updated automatically and all file items are evenly spaced in the .TEXT file.

When editing program text, the user is free to split rules apart or add new rules before and after the rules being edited.

Users are responsible for saving changes from the text editor before exiting. Most text editors do not update files edited until the user asks for an update. This update must be performed or ROSIE will ask the user if he wishes to abort the editing session. If the session is aborted, no changes are made to the program file. If not, ROSIE attempts to return to the edit session (which may not work with some text editors).

NOTE: If you select an editor which associates line numbers with each line (e.g. SOS or DEC's EDIT), you must leave each edit session with a command that does not store the line numbers in the file written by the editor (U for SOS, EU for EDIT). If line numbers are left in, an INTERLISP error will occur.

show fileterm -- displays filesegment text to the terminal.

trace [fileterm] -- the trace facility can be used to monitor the invocation of rulesets. When a traced ruleset is invoked, a message is printed and the values of its arguments are given. When the ruleset has finished executing, another message is printed and the value being returned (if any) is given. **Trace** alone traces all currently defined rulesets. **Trace** fileterm, where fileterm is the name of a file, traces all rulesets in that file.

untrace [fileterm] -- untrace the ruleset fileterm, all the rulesets in the file named fileterm, or all rulesets.

NOTE: Attempting to trace a single rule in a file or ruleset will result in a run-time error.

erase fileterm -- removes the filesegment from its file and discards it. Asks for confirmation from the user before making the change.

NOTE: Fileterms that are rulesets will still be defined during the session even after they have been erased. Thus, removing the predicate "is married" from the file "foo" will not cause subsequent invocations of "is married" to bomb. At present there is effectively no way to rid yourself of a ruleset once it has been noticed short of erasing it from the file, logging out of ROSIE, and starting a new session.

copy fileterm **to** (¦ **before** ¦ **after** ¦) fileterm -- copies one filesegment to before or after another. The file that receives the copied text is rewritten. Some intuitive restrictions are placed on the kinds of changes that can be made.

move fileterm **to** (¦ **before** ¦ **after** ¦) fileterm -- similar to **copy**, but erases the original filesegment from its file after copying it. Both files involved are rewritten (or only one is rewritten if the move is made within a single program file).

insert (¦ **before** ¦ **after** ¦) fileterm -- invokes the user's text editor, allowing the user to compose program text which is then parsed and inserted before or after the indicated filesegment. The file is then rewritten. **Insert** can be used to add text to the end of a program file without requiring any existing text to be reparsed.

edit fileterm -- invokes the user's text editor to edit the indicated filesegment text. The new text is then parsed and the old filesegment replaced. NOTE: to edit just one rule in a file you must use the form **edit** filename rulenumber.

find string **in** fileterm -- displays the lines of a program file that contains the string. Upper- and lower-case characters in string are equivalent. The location of these lines is indicated in terms of the rulesets or rules which contain them. This is very useful for finding things when programs get large.

scan fileterm -- this feature outlines the filesegment by naming rulesets, indicating compile status, pinpointing syntax errors, etc. For example, **scan** program will outline the contents of file PROGRAM.

7. USER AIDS

7.1. OVERVIEW

The purpose of this chapter is to provide the user with information that should be helpful in using the ROSIE system and debugging ROSIE programs.

7.2. ERRORS

Errors cause ROSIE to abort ruleset execution and return to the top level. The error message printed provides an indication of the problem. Below, we provide most of the error messages ROSIE produces, arranged alphabetically. The number in parentheses is the section of this manual that may pertain. Text following ">>>" is an attempt to help you diagnose the problem.

Attempt to assert or deny EQUALITY. (4.6.3)
Attempt to assert or deny GREATER THAN. (4.6.3)
Attempt to assert or deny GREATER THAN OR EQUAL TO. (4.6.3)
Attempt to assert or deny IS MATCHED BY. (4.6.4)
Attempt to assert or deny LESS THAN. (4.6.3)
Attempt to assert or deny LESS THAN OR EQUAL TO. (4.6.3)
Attempt to assert or deny THERE IS. (4.6.4)
>>>Any sentence that contains a non-primitive verb phrase cannot be asserted or denied.
Attempt to create illegal NAME from the string (2.3.2)
Attempt to create illegal NUMBER from the string (2.3.4)
Attempt to create NUMBER with illegal UNITS from the string (2.3.4)

Bad file name (6.7)
Bad PROCEDURE name in CALL (5.4)
Body is not a SPREAD-NLAMBDA expression. (5.11)
>>>System ruleset definition was not SPREAD-NLAMBDA.
BUILD can't create filepackage files for (6.7)
>>>Check to see if the file already exists, is protected, or no disk space.
BUILD requires an atomic file name. (6.7)

Can't change file loaded from another directory (6.7)
Can't COPY ruleset rules to a point outside any ruleset (6.7)
Can't COPY rulesets or file rules to within a ruleset (6.7)

Can't create an edit file! (6.7)
   >>>Check to see if file is in another directory, no disk space, file is
      protected, or file doesn't exist.
Can't create file (6.7)
   >>>No disk space, file already exists, etc.
Can't delete file (6.7)
   >>>File is protected against deletion or in another directory.
Can't find file (6.2)
Can't find filepackage files for (6.2)
Can't find PARSE file for (6.2)
Can't find SAVE file (5.13)
Can't find SUCH reference (4.5)
Can't find TEXT file (6.2)
Can't find THAT reference (4.2.8)
   >>>Attempting to resolve anaphora with no antecedent description.
Can't find this file segment (6.6)
Can't MOVE ruleset rules to a point outside any ruleset (6.7)
Can't MOVE rulesets or file rules to within a ruleset (6.7)
Can't rename file (6.7)
CHANGE requires atomic filenames (6.7)
COMPILE requires an atomic filename (6.2)
Computation depth limit exceeded
   >>>Check for infinite recursive loops.
CONCLUDE not inside a PREDICATE. (5.6)

Device not OPEN (5.13)
Device not open for input (5.13)
Device not OPEN for output (5.13)
Discarding a spurious PRIVATE or MONITOR declaration (5.2,5.10)
   >>>Only one private or monitor statement allowed per ruleset.
Discarding an unexpected END (5.3)
Discarding an unexpected RULESET header (5.3)
DIRECTORY takes an atom or string (6.7)
Does not describe a known file or ruleset (6.7)

File already exists (6.2)
File no longer known to file package (6.2)
Filename not known to file package (6.2)
Filepackage files already exist named (6.2)
File still contains syntax errors (6.7)
   >>>You cannot compile a file that has syntax errors.
File is already compiled (6.7)

Illegal comparison (2.3.4)
   >>>Attempting to compare non-comparable numbers.

Illegal DATABASE file (3.1)
Illegal DATABASE name (3.1)
Illegal DRIBBLE file (5.13)
Illegal file name in LOAD (6.2)
Illegal FILESEGMENT descriptor(6.6)
Illegal operation (4.6.3)
   >>>Arithmetic operations require consistent labels or units.
Illegal LET action; one TERM must be THE DESC or TERM'S DESC (3.7)
Illegal PORT name (5.13)
Illegal UNITS following a number (2.3.4)
Improper SAVE file name (5.13)
Input from device won't match pattern (4.9)
Input from terminal won't match pattern (4.9)
Input is ambiguous at (4.2)

Line contains unterminated comment (2.2.2)

MISSING END. (5.3)
MONITOR declaration doesn't belong here (5.10)

No DRIBBLE file currently active (5.13)
No such element exists (3.3)
   >>>"the" requires the existence of at least one (and preferably only
      one) such element.
No such DATABASE file (3.7)
Not a FILESEGMENT element (6.6)
Not a proposition (4.6.2)
Nothing saved for line (7.4)
   >>>Only the last 40 lines are remembered.

Old COMPILE file has been deleted! File must be reloaded (6.2)
Only RULES can be executed here (B.2)
   >>>You are at the top level of ROSIE. You must build or edit a file
      to create a ruleset.

PARSE requires an atomic file name (6.2)
PARSE can't find file (6.2)
PARSE can't create parse files for (6.2)
PARSE can't create filepackage files for (6.2)
   >>>All of the above indicate an improper file name.
      Consult the TOPS-20 documentation.

PORT already open (5.13)

PRIVATE declaration doesn't belong here (5.8)
Procedure not defined (5.4)
PRODUCE not inside a GENERATOR (5.5)

Quota exceeded or disk full...
   >>>If you received this error message it is because you have used up all your directory space. You have also been dumped into the EXEC. Unless you have crucial files deleted, just type "expunge" and, after the pages have been freed, "continue." This will leave you back in ROSIE right where you were.
RETURN not inside a RULESET (5.3)
Ruleset no longer known to file package (B.2)
Ruleset not known to file package (B.2)

Same preposition used twice (3.1)
   >>>Each preposition introduces an element; that element will be associated with the preceding object. NO object may have the same preposition used twice.
Sections of rulesets are not permitted (6.7)
   >>>You cannot trace or untrace single rules.
Segments overlap in a MOVE statement (6.7)
Some definitions in-core are not from this file! file must be reparsed(6.2)
Syntax error at (6.2)
Syntax error is (6.2)
SYSTEM GENERATOR returned illegal value (5.11)
   >>>Rewrite the system generator to produce an appropriate ROSIE element.

THAT refers to an unbound local variable.
   >>>A sentence relying on a construct like
         "any...which...any x...that x..."
      causes real problems, since ROSIE cannot resolve the actual binding at the time the sentence is used. Give up and rewrite the sentence.
Too many PRIVATE declarations (5.2)
Too many MONITOR declarations (5.10)
   >>>Only one declaration of monitor type is allowed. Similarly for private variables.

Unknown INFO request (7.3)
Unbound local variable (5.2)
Unmatched left paren or missing body (5.11)
   >>>A system generator definition either is missing or was not closed correctly.

7.3. USER SUPPORT ACTIONS

The following actions provide general support for user interaction. The **fix** and **redo** actions deal with rules executed at the top level by the user. ROSIE remembers the last 40 rules typed. The **info** action answers some global questions about memory space, program files, etc.

fix linenumber -- allows users to edit and resubmit rules typed to the top level. Invokes the user's text editor on the rule indicated and executes the modified rule when the editor is exited. The modified rule is remembered instead of the **fix** action that exhumed it, so it can be referred to later by its line number.

redo linenumber [**thru** linenumber] [integer **times**] -- reexecutes one or more rules typed to the top level. More than one rule can be reexecuted by using **redo** linenumber **thru** linenumber. The rule or rule sequence can be reexecuted more than once by supplying the integer **times** argument.

info storage -- prints a rather verbose accounting of memory space after collecting unused space (garbage collection). The most informative part of the printout for most users is at the bottom and tells how many pages of memory are left. Since this action forces a garbage collection, it can be executed just before the **save** action to clean things up a bit.

info files -- enumerates all filepackage program files in the user's directory, along with date last modified, size of the text file, compile status, and whether that file has been loaded.

info loaded -- tells which program files have been loaded, including those loaded from other directories. Also gives date of last modification, size of text file, compile status.

info users -- tells which users are currently on the computer.

info date -- prints the date, time, and name of the program in comment form.

7.4. ROSIE'S TOP LEVEL AND USER INTERACTION

When ROSIE is entered, a user interacts with the system by typing single rules at the terminal. Each rule typed is immediately executed, after which the user is prompted for another rule. This cycle continues until the ROSIE session is terminated. When waiting for rules to be typed, ROSIE is said

to be at the top level.

At the top level, ROSIE prompts for each rule with a line number. Thereafter this number can be used to refer to the line typed. Lines typed are remembered by ROSIE for future reference and can be edited, resubmitted, and displayed (see section 7.3).

Sample session:

```
            [ ROSIE  Friday, January 30, 1981  8:50pm ]

<1> assert john is a man.
<2> display every man.
JOHN
<3> assert each of sam, bill and joe is a man.
<4> display every man.
JOHN
SAM
BILL
JOE
<5> assert mary does like every man.
<6> ?
[ Global Database ]
    MARY does like JOHN.
    MARY does like SAM.
    MARY does like BILL.
    MARY does like JOE.
    JOHN is a man.
    SAM is a man.
    BILL is a man.
    JOE is a man.

<7> logout.
```

Note in the above example that line numbers are displayed in angle brackets "<>" before each rule. In this example, the user is interactively building a simple database, which he displays just before exiting ROSIE.

Useful commands related to user interaction:

Line history -- ROSIE remembers up to 40 lines typed by the user, forgetting the oldest lines as new ones are typed. Actions exist which allow the user to edit these lines to be resubmitted (fix) or to resubmit one or more lines as typed (redo) (see section 7.3). Once a line is forgotten, it can no longer be referenced by these actions.

Interrupt -- Control-B is the ROSIE Interrupt character. When typed, it causes control to return to the top, regardless of what is being executed. It will also erase a partially typed rule being submitted and repeat the line number prompt. This provides a clean way to terminate program execution but may leave programs in a funny state, since effects of execution up to the interrupt cannot be reversed.

Inform -- Control-T is the ROSIE Inform character. When typed, the user is told what ROSIE is doing (which ruleset is running, which rule is being executed). Program execution is not affected.

Editing -- Some actions cause the user's text editor to be invoked. ROSIE always creates a temporary file for editing applications; the name of the file is passed to the editor. Users are responsible for saving the results of editing sessions and exiting normally from the editor. When the editor is exited without having updated the temporary file, ROSIE will give the user a chance to return to the editing session; otherwise the edit will be aborted and ROSIE will return control to the top level. In the TOPS-20 implementation, ROSIE will invoke the editor defined by the logical name "Editor:" when a text editor is required. Users who do not use the default editor can redefine Editor: by typing the following line to his EXEC:

```
DEFINE EDITOR: youreditor
```

If your editor asks for a filename explicitly, rather than being able to read one from the TOPS-20 command line, you should type ROSIE-EDITOR.

A few special rules designed to improve user interaction are included as legal rules in the language. These rules are legal in rulesets, but are primarily useful while interacting at the top level:

?? -- displays the last 40 lines typed to top level.

? -- displays the current contents of the global database.

term ? -- displays all relationships in the global database that include the element value of the term.

primitive sentence ? -- displays all relationships in the global database built from the same relational form that have equal elements.

Sample session:

[ROSIE Sunday, February 1, 1981 7:37pm]

```
<1> Assert each of Bill, Sam, Dick and Henry is a boy.
<2> Assert each of Mary, Sue, Jane and Margaret is a girl.
<3> Assert Bill does like Margaret and Sue does like Henry.
<4> Assert any boy is a person and any girl is a person.
<5> ??

<1> ASSERT EACH OF BILL, SAM, DICK AND HENRY IS A BOY.
<2> ASSERT EACH OF MARY, SUE, JANE AND MARGARET IS A GIRL.
<3> ASSERT BILL DOES LIKE MARGARET AND SUE DOES LIKE HENRY.
<4> ASSERT ANY BOY IS A PERSON AND ANY GIRL IS A PERSON.
<5> ??

<6> ?
[ Global Database ]
    BILL does like MARGARET.
    SUE does like HENRY.
    BILL is a boy.
    SAM is a boy.
    DICK is a boy.
    HENRY is a boy.
    MARY is a girl.
    SUE is a girl.
    JANE is a girl.
    MARGARET is a girl.
    ANY BOY is a person.
    ANY GIRL is a person.

<7> Bill?
[ BILL ]
    BILL does like MARGARET.
    BILL is a boy.

<8> Which boy does like Margaret?

    BILL does like MARGARET.
<9> Which person does like which person?

    BILL does like MARGARET.
```

```
    SUE does like HENRY.
<10> logout.
```

7.5. ROSIE BNF

A BNF description of ROSIE'S syntax is given for the user who wants a terse but complete definition of the language. The following conventions apply:

- uppercase (without a "$") indicates that the word appears on the left-hand side of some production; lowercase indicates that the word is recognized by ROSIE.
- the use of the "$" character, i.e., in $ATOM, $NUMBER, and $STRING, indicates a placeholder for a word of the given type. $ATOM requires a ROSIE name (2.3.2); $NUMBER, and $STRING a ROSIE number (2.3.4) and ROSIE string (2.3.3), respectively.
- the use of the word Nil indicates that the production need not decompose further.
- comments are surrounded by "***"

```
<PROGRAM> ::= <DECLARATION> <RULES> <ENDBLOCK> ¦ <ONERULE>
     ¦ <QUERY>

<DECLARATION> ::= <HEADER>
                  <HEADER> <PRIVATEDEC> ¦
                  <HEADER> <MONITORDEC> ¦
                  <HEADER> <PRIVATEDEC> <MONITORDEC>


  <RULES> ::= <ONERULE> <RULES> ¦ <ONERULE>

  <ENDBLOCK> ::= end .

  <PRIVATEDEC> ::= private <PRIVATELIST> .

    <PRIVATELIST> ::= $ATOM , <PRIVATELIST> ¦ $ATOM

  <MONITORDEC> ::= execute <MONTYPE>

    <MONTYPE> ::= sequentially ¦ randomly ¦ cyclically
```

```
<HEADER> ::= to generate <FUNCTION-FORM> :                          ¦
             system ruleset to generate <FUNCTION-FORM> :           ¦
             to <PROCEDURE-FORM> :                                  ¦
             system ruleset to <PROCEDURE-FORM> :                   ¦
             to decide <PREDICATE-FORM> :                           ¦
             system ruleset to decide <PREDICATE-FORM> :

  <FUNCTION-FORM> ::= $ATOM <PRIVATEPP>            ***Ruleset
                                                      Forms***
  <PROCEDURE-FORM> ::= $ATOM <PRIVATEPP> ¦ $ATOM $ATOM
       <PRIVATEPP>

  <PREDICATE-FORM> ::= $ATOM <ISF*> <AAN> $ATOM <PRIVATEPP>  ¦
                       $ATOM <ISF*> $ATOM <PRIVATEPP>        ¦
                       $ATOM <ISF*> $ATOM $ATOM <PRIVATEPP>  ¦
                       $ATOM <DOESF*> $ATOM <PRIVATEPP>      ¦
                       $ATOM <DOESF*> $ATOM $ATOM <PRIVATEPP>

    <AAN> ::= a ¦ an

    <ISF*> ::= was      ¦ was not  ¦
               is       ¦ is not   ¦
               were     ¦ were not ¦
               am       ¦ am not   ¦
               are      ¦ are not  ¦
               will be  ¦ will not be

    <DOESF*> ::= did  ¦ did not  ¦
                 do   ¦ do not   ¦
                 does ¦ does not ¦
                 will ¦ will not


    <PRIVATEPP> ::= Nil ¦ <P1>

     <P1> ::= <P1> <PREP> $ATOM ¦ <PREP> $ATOM


<QUERY> ::= ? ¦ ?? ¦ <TERM> ? ¦ <PRIMITIVE-SENTENCE> ?

<ONERULE> ::= <ACTIONBLOCK> . ¦ <ACTIONBLOCK> !

  <ACTIONBLOCK> ::= <ACTLIST>
```

```
<ACTLIST> ::= <ACT> ¦ <ACT> and <ACTLIST>

<COLONBLOCK> ::= <ACTIONBLOCK> ; ¦ <ACTIONBLOCK>

<COMMABLOCK> ::= <ACTLIST> , ¦ <ACTLIST>

<ACTION> ::= <ACT>

<ACT> ::= ( <ACTLIST> ) ¦                    ***Multiple Action
          <ACT> , <COMMABLOCK>                  Blocks***


<ACT> ::=
     assert <ANDFORM>      ¦                  ***Database
     deny <ANDFORM>        ¦                     Actions***
     assert <ANDTERM>      ¦
     deny <ANDTERM>        ¦
     <LET-ACTION>          ¦
     create <AAN> <DESC>   ¦
     describe <TERM>       ¦
     forget about <TERM>   ¦
     dump                  ¦
     activate <TERM>       ¦
     activate              ¦
     clear database        ¦
     deactivate            ¦
     dump as <TERM>        ¦
     restore <TERM>


<ANDFORM> ::= <SENTENCE> ¦ <ANDFORM> and <SENTENCE>

<ANDTERM> ::= <TERM> ¦ <ANDTERM> and <TERM>

<LET-ACTION> ::= let <LETFORM> ¦ <LET-ACTION>
     and <LETFORM>

         <LETFORM> ::= <TERM> be <TERM>


                                          ***Flow-of-Control
                                             Actions***
<ACT> ::=
     <ITERATE>                                          ¦
     <IFACT>                                            ¦
     match <TERM> against <PATTERN>                     ¦
     select <TERM> : <SELECTBLOCK> <DEFAULTBLOCK>       ¦
     match <TERM>  : <MATCHBLOCK> <DEFAULTBLOCK>        ¦
     choose situation : <CHOOSEBLOCK> <DEFAULTBLOCK>    ¦
```

```
   go $ATOM <OPTPPHRASE>                                    ¦
   go $ATOM <TERM> <OPTPPHRASE>                             ¦
   call <TERM> <OPTPPHRASE>                                 ¦
   call <TERM> (using TERM) <OPTPPHRASE>                    ¦
   do nothing                                               ¦
   return                                                   ¦
   produce <TERM>                                           ¦
   conclude true                                            ¦
   conclude false                                           ¦
   quit                                                     ¦
   quit because <PATTERN>                                   ¦
   wait for $NUMBER seconds                                 ¦
   save as <TERM>                                           ¦
   revert to <TERM>

<ITERATE> ::=
   for each <DESC> while <CONDITION> until <CONDITION>
        <ACTION>                                                ¦
   for each <DESC> while <CONDITION> <ACTION>                   ¦
   for each <DESC> until <CONDITION> <ACTION>                   ¦
   while <CONDITION> until <CONDITION> <ACTION>                 ¦
   for each <DESC> <ACTION>                                     ¦
   while <CONDITION> <ACTION>                                   ¦
   until <CONDITION> <ACTION>                                   ¦

<IFACT> ::=
   if <CONDITION> <ACTION>                          ¦
   unless <CONDITION> <ACTION>                      ¦
   if <CONDITION> <ACTION> otherwise <ACTION>       ¦
   unless <CONDITION> <ACTION> otherwise <ACTION>

<SELECTBLOCK> ::= <TUPLE1> <COLONBLOCK>             ¦
                  <SELECTBLOCK> <TUPLE1> <COLONBLOCK>

<MATCHBLOCK> ::= <PATTERN> <COLONBLOCK>             ¦
                 <MATCHBLOCK> <PATTERN> <COLONBLOCK>

<CHOOSEBLOCK> ::= if <CONDITION> <COLONBLOCK>       ¦
                  <CHOOSEBLOCK> if <CONDITION>
                       <COLONBLOCK>

<DEFAULTBLOCK> ::= Nil ¦ default : <COLONBLOCK>


<ACT> ::=
     display <TERM>                 ¦        ***Input/Output
     open <TERM> to read            ¦           Actions***
```

```
    open <TERM> to write
    open <TERM> to append
    open port to <TERM>
    close <TERM>
    send <PATTERN>
    send to <TERM> <PATTERN>
    read <PATTERN>
    read <TIMEOUT> <PATTERN>
    dribble to <TERM>
    stop dribbling
    read from <TERM> <PATTERN>
    read <TIMEOUT> from <TERM> <PATTERN>

  <TIMEOUT> ::= for $NUMBER seconds


<ACT> ::=
    dir                          ***File
    dir <TERM>                      Actions***
    type <TERM>
    delete <TERM>
    copy <TERM> to <TERM>
    append <TERM> to <TERM>
    rename <TERM> to TERM


<ACT> ::=
    parse <TERM>                 ***File Package
    load <TERM>                     Actions***
    sysload <TERM>
    compile <TERM>
    build <TERM>
    info <TERM>
    change <TERM> to <TERM>


<ACT> ::=
    show <FPTERM>                ***File Package
    scan <FPTERM>                   Filesegment
    erase <FPTERM>                  Actions***
    edit <FPTERM>
    trace
    untrace
    trace <FPTERM>
    untrace <FPTERM>             (NOTE: attempting
    find $STRING in <FPTERM>      to TRACE a single
    copy <FPTERM> to before <FPTERM>  rule will result
```

```
    copy <FPTERM> to after <FPTERM>  ¦     in a run-time error)
    move <FPTERM> to before <FPTERM> ¦
    move <FPTERM> to after <FPTERM>  ¦
    insert before <FPTERM>           ¦
    insert after <FPTERM>


  <FPTERM> ::=
      " <FILESEG> "                  ¦
      $ATOM                          ¦
      $ATOM , $INTEGER               ¦
      $ATOM , $INTEGER $INTEGER      ¦
      file $ATOM <RULE-SPEC>


   <FILESEG> ::=
        to generate <FUNCTION-FORM> <RULE-SPEC> ¦
        to <PROCEDURE-FORM> <RULE-SPEC>         ¦
        to decide <PREDICATE-FORM> <RULE-SPEC>


   <RULE-SPEC> ::=
        Nil        ¦
        , $INTEGER ¦
        , $INTEGER $INTEGER

<ACT> ::=
     fix $INTEGER                    ¦      ***Line
     redo $INTEGER                   ¦         History
     redo $INTEGER thru $INTEGER     ¦         Actions***
     redo $INTEGER $INTEGER times    ¦
     redo $INTEGER thru $INTEGER $INTEGER times


<ACT> ::=
     push ¦                                 ***System
     logout                                    Support
                                               Actions***


  <CONDITION> ::= <PRIMITIVE-CONDITION>

    <PRIMITIVE-CONDITION> ::=
                <COND>                                ¦
                <PRIMITIVE-CONDITION> , and <COND>    ¦
                <PRIMITIVE-CONDITION> , or <COND>

      <COND> ::= <COND> or <COND-CONJUNCT> ¦
```

```
                    <COND-CONJUNCT>

          <COND-CONJUNCT> ::= <COND-CONJUNCT> and
                    <COND-PRIMARY> ¦
                              <COND-PRIMARY>

            <COND-PRIMARY> ::= ( <COND> ) ¦ <BASE-SENTENCE>



<PRIMITIVE-SENTENCE> ::= <TERM> <PRIMITIVE-FORM>

<SENTENCE> ::= ( <SENTENCE> ) ¦ <BASE-SENTENCE>

<BASE-SENTENCE> ::= <TERM> <VERB-PHRASE> ¦ there is <HOWMANY>
      <DESC>

<HOWMANY> ::= no ¦ <AAN> ¦ just one ¦ more than one


<TERM> ::= <SUBTERM> ¦ <EXPRESSION>

  <SUBTERM> ::= $STRING                          ¦
                <NAMELIST>                       ¦
                $NUMBER                          ¦
                $NUMBER <NAMELIST>               ¦
                <NAMELIST> $NUMBER               ¦
                <TUPLE1>                         ¦
                <TUPLE2>                         ¦
                ` <PRIMITIVE-SENTENCE> '         ¦
                the <DESC>                       ¦
                <TERM> 's <DESC>                 ¦
                that $ATOM                       ¦
                <AAN> <DESC>                     ¦
                <AAN> new <DESC>                 ¦
                some <DESC>                      ¦
                every <DESC>                     ¦
                each of <TERMLIST> and <TERM>    ¦
                each of <TERMLIST>, and <TERM>   ¦
                one of <TERMLIST> or <TERM>      ¦
                any <DESC>                       ¦
                one of ,TERMLIST>, and <TERM>    ¦
                which <DESC>                     ¦
                the string <PATTERN>             ¦
                the name <PATTERN>               ¦
                the number <PATTERN>             ¦
```

```
<NAMELIST> ::= $ATOM ¦ <NAMELIST> $ATOM <NAMELIST>

  <TUPLE1> ::= < > ¦ < <TERMLIST> >

  <TUPLE2> ::= the tuple containing each <DESC>

  <TERMLIST> ::= <TERM> ¦ <TERM> , <TERMLIST>


<EXPRESSION> ::= <LEVEL2>
                 <EXPRESSION> + <LEVEL2> ¦
                 <EXPRESSION> - <LEVEL2> ¦

  <LEVEL2> ::= <LEVEL1>            ¦
               <LEVEL2> * <LEVEL1> ¦
               <LEVEL2> / <LEVEL1>

    <LEVEL1> ::= <PRIMARY-EXPR> ¦
                 <PRIMARY-EXPR> ** <LEVEL1>

      <PRIMARY-EXPR> ::= ( <TERM> ) ¦ <SUBTERM>


<DESC> ::= such $ATOM ¦ <ADJECTIVES> <DESC-CLASS> <RELCLAUSE>

  <DESC-CLASS> ::= $ATOM <OPTVAR> <OPTPPHRASE>

  <ADJECTIVES> ::= <ADJECTIVE-LIST>

   <ADJECTIVE-LIST> ::= <ADJECTIVE> ¦ <ADJECTIVE> <ADJECTIVE-
        LIST>

     <ADJECTIVE> ::= $ATOM

  <OPTVAR> ::= Nil ¦ ( $ATOM )

  <OPTPPHRASE> ::= Nil ¦ <PPHRASE>


<PPHRASE> ::= <PPHRASE> <PP> ¦ <PP>

  <PP> ::= <PREP> <TERM> ¦ ( <PREP> <TERM> )

     <PREP> ::= above ¦ at ¦ for  ¦ outside ¦ without ¦
                after ¦ of ¦ per  ¦ during  ¦ because ¦
                among ¦ by ¦ with ¦ against ¦ behind  ¦
```

```
                along ¦ as ¦ from ¦ around  ¦ before  ¦
                below ¦ in ¦ near ¦ beside  ¦ inside  ¦
                while ¦ up ¦ over ¦ within  ¦ through ¦
                until ¦ on ¦ into ¦ across  ¦ toward  ¦
                since ¦ to ¦ onto ¦ under   ¦ about
```

```
<VERB-PHRASE> ::= <PRIMITIVE-VERBPHRASE> ¦ <OTHER-VERBPHRASES>

  <PRIMITIVE-VERBPHRASE> ::= <PRIMITIVE-FORM>

  <OTHER-VERBPHRASES> ::=
              is <AAN> <DESC>                ¦
              is not <AAN> <DESC>            ¦
              is provably true               ¦
              is provably false              ¦
              is not provably true           ¦
              is not provably false          ¦
              is matched by <PATTERN>        ¦
              is not matched by <PATTERN>    ¦
              has <HOWMANY> <DESC>           ¦
              <RELATIONAL-OPERATOR> <TERM>


    <RELATIONAL-OPERATOR> ::=
              is greater than                   ¦ >      ¦
              is greater than or equal to       ¦ > =    ¦
              is less than                      ¦ <      ¦
              is less than or equal to          ¦ < =    ¦
              is equal to                       ¦ =      ¦
              is not equal to                   ¦ ~ =    ¦
              is not greater than               ¦ ~ >    ¦
              is not less than                  ¦ ~ <    ¦
              is not greater than or equal to   ¦ ~ > =  ¦
              is not less than or equal to      ¦ ~ < =

<PRIMITIVE-FORM> ::=
          <ISF> <AAN> $ATOM <OPTPPHRASE>   ¦
          <ISF> $ATOM <OPTPPHRASE>         ¦
          <ISF> $ATOM <TERM> <OPTPPHRASE>  ¦
          <DOESF> $ATOM <OPTPPHRASE>       ¦
          <DOESF> $ATOM <TERM> <OPTPPHRASE>

  <ISF> ::= was      ¦ was not  ¦
            is       ¦ is not   ¦
            were     ¦ were not ¦
            am       ¦ am not   ¦
```

```
            are      ¦ are not  ¦
            will be  ¦ will not be

  <DOESF> ::= did   ¦ did not  ¦
              do    ¦ do not   ¦
              does  ¦ does not ¦
              will  ¦ will not


<RELCLAUSE> ::= <RC1>

  <RC1> ::= <RC> ¦ <RC1> and <RC> ¦ <RC1> or <RC>

    <RC> ::= ( <RC> )                                                    ¦
             <SUCH-THAT> <SENTENCE>                                      ¦
             <SUCH-THAT> ( <PRIMITIVE-CONDITION> )                       ¦
             <THAT-WHO-WHICH> <VERB-PHRASE>                              ¦
             <THAT-WHO-WHICH> <TERM> <ISF> $ATOM <OPTPPHRASE>            ¦
             <THAT-WHO-WHICH> <TERM> <DOESF> $ATOM <OPTPPHRASE>          ¦
             <THAT-WHO-WHICH> <TERM> <RELATIONAL-OPERATOR>               ¦
             <PREP> <WHICH-WHOM> <TERM> <PRIMITIVE-FORM>                 ¦
             whose <DESC> is $ATOM
             whose <DESC> is not $ATOM

      <SUCH-THAT> ::= where ¦ such that

      <THAT-WHO-WHICH> ::= that ¦ who ¦ which

      <WHICH-WHOM> ::= which ¦ whom


<PATTERN> ::= { <PATTERNS-LIST> }


  <PATTERNS-LIST> ::= <PATTERN-SPECIFICATION-LIST> ¦
                      <PATTERN-SPECIFICATION-LIST> ,
                           <PATTERNS-LIST>

    <PATTERN-SPECIFICATION-LIST> ::=
             <SPECIFICATION1>                                            ¦
             <SPECIFICATION1> ( bind $ATOM )                             ¦
             <SPECIFICATION1> ( bind $ATOM to the string )               ¦
             <SPECIFICATION1> ( bind $ATOM to the name )                 ¦
             <SPECIFICATION1> ( bind $ATOM to the number )               ¦

      <SPECIFICATION1> ::= <PATTERN-SPECIFICATION> ¦ <TERM>
```

```
<PATTERN-SPECIFICATION> ::=
     <SPEC-QUANTITY> line                                          ¦
     <SPEC-QUANTITY> lines                                         ¦
     <SPEC-QUANTITY> <SPEC-RESTRICTION>                            ¦
     <SPEC-QUANTITY> <SPEC-RESTRICTION> in $STRING                 ¦
     <SPEC-QUANTITY> <SPEC-RESTRICTION> not in $STRING             ¦
     anything                                                      ¦
     something                                                     ¦
     control $STRING                                               ¦
     quote                                                         ¦
     codes ( NUMLIST )                                             ¦
     one of ( <PATTERNS-LIST> )                                    ¦
     each of ( <PATTERNS-LIST> )                                   ¦
     { <PATTERNS-LIST> }                                           ¦
     return                                                        ¦
     end                                                           ¦

  <SPEC-QUANTITY> ::=
           $INTEGER          ¦
           $INTEGER or more  ¦
           $INTEGER or less

  <SPEC-RESTRICTION> ::=
            letters          ¦
            nonletters       ¦
            numbers          ¦
            nonnumbers       ¦
            alphanumerics    ¦
            nonalphanumerics ¦
            blanks           ¦
            nonblanks        ¦
            controls         ¦
            noncontrols      ¦
            characters

  <NUMLIST> ::= $INTEGER ¦ $INTEGER , <NUMLIST>
```

APPENDIX A

GETTING STARTED WITH ROSIE

The following sample session is provided to give the new user a feeling for programming in the ROSIE environment. This demonstration is by no means complete but should make the reference manual easier to understand and your first attempt at interacting with ROSIE more successful.

ROSIE produced the text beginning on unnumbered, unindented lines. Lines beginning with a bracketed number were typed by hand. Line <1> below is missing because it commanded ROSIE to **dribble** this session to a file.

```
                TM
        [ Rosie   Friday, August 7, 1981  1:18pm ]

<2>  assert john is a man.
<3>  assert each of mary and sara is a woman.
<4>  ?
[ Global Database ]
    JOHN is a man.
    MARY is a woman.
    SARA is a woman.
```

> Programming in ROSIE consists primarily of actions performed on information in the database. The database is built up by asserting ROSIE sentences. Lines <2> and <3> above are examples of assertions. In line <2> we tell ROSIE that "john" is an element contained in the class "man". In line <3> we see how to make the same sort of assertion for more than one class member. The assertion "assert mary is a woman and sara is a woman." is equivalent to <3>. In line <4> we use the character "?" to look at the contents of the database. Notice how the compound assertion (<3>) has been broken into two database sentences. Notice also how ROSIE capitalizes all elements (the words "man" and "woman" are not elements in these sentences--they are part of the relations "is-a-man" and "is-a-woman" and are called "relation-names").

```
<5>  assert any man does like any woman.
<6>  ?
```

```
[ Global Database ]
    ANY MAN does like ANY WOMAN.
    JOHN is a man.
    MARY is a woman.
    SARA is a woman.
```

Lines <5> and <6> show us another kind of element: the "class element". Here, "ANY MAN" and "ANY WOMAN" are elements that describe a class of elements. Thus, the element "ANY MAN" stands for every element in the database satisfying the "is-a-man" relation. Notice that ROSIE requires the form "does like" instead of simply "like".

```
<7>  display every woman that john does like.
MARY
SARA
```

In order to determine which women "john does like", ROSIE scans the database for the "does-like" relation. What ROSIE finds is the relationship "ANY MAN does like ANY WOMAN." Because ROSIE is generating elements from the database, and because this relationship uses at least one class element, ROSIE must generate the set of relationships that "ANY MAN does like ANY WOMAN" represents. Generating sentences means substituting each element in the class specified by the any for the class elements in the relationship. Because we asked for "every woman", ROSIE will generate all the relationships that result from substituting each element in the class of "man" for "ANY MAN" and each element in the class of "woman" for "ANY WOMAN". In terms of the database displayed by <6>, this means ROSIE generates the relationships "john does like mary" and "john does like sara". In the process of generating these, ROSIE checks each one to see if it meets the criteria stated in <7>. Note that if we had asked for "a woman that john does like" or "some woman that john does like", ROSIE would have stopped as soon as it had generated one sentence that met the criteria.

```
<8>  deny any man does like any woman.
<9>  ?
```

```
[ Global Database ]
    JOHN is a man.
    MARY is a woman.
    SARA is a woman.
```

In <8> we have an example of how to delete a sentence from the database. Notice that the generation of sentences caused by <7> did not assert anything into the database; indeed, had we retyped <7> instead of <8>, ROSIE would have had to compute the response all over again. Note also that denial of the relationship containing an **any** construct requires that the **any** construct be typed exactly as it appears in the relationship. This is especially important when the **any** contains adjectives or relative clauses, i.e., "deny any big man..." is not equivalent to "deny any man who is big...".

```
<10>  assert any man does like a woman.
<11>  ?
[ Global Database ]
    ANY MAN does like MARY.
    JOHN is a man.
    MARY is a woman.
    SARA is a woman.
```

In <10> we see an important difference between **any** and **a**. The article **a** is not a class element and, therefore, does not "delay evaluation". Instead, ROSIE picks some element satisfying the criteria "is-a-woman" and substitutes it in. The choice of MARY is arbitrary. The semantics of **a** take some time to master but a thorough knowledge of its actions is important if you want to avoid undesirable side-effects.

```
<12>  forget about mary and forget about sara.
<13>  ?
[ Global Database ]
    JOHN is a man.
```

Here is another example of how to delete information from the database. "Forget about x" where x is an

element or evaluates to an element causes every sentence containing x to be deleted from the database.

```
<14>  assert any man does like a woman.
<15>  ?
[ Global Database ]
    ANY MAN does like WOMAN #1.
    JOHN is a man.
    WOMAN #1 is a woman.
```

Here we find out still more about the semantics of **a**. When there is no element in the database that satisfies the "is-a-woman" relation, the use of **a** demands that one be created. After "WOMAN #1 is a woman." is inserted, the reference to "a woman" evaluates to "WOMAN #1".

```
<16>  display the woman.
WOMAN #1
```

The other article, **the**, always refers to the one and only element that satisfies the specified relation. If there is more than one element that does so, one is chosen at random. However, if the article **the** is used in a **let** action, e.g., "let the woman be mary.", the effect would be to deny all the "is-a-woman" relations already in the database and then assert the sentence "mary is a woman".

```
<17>  clear database.
<18>  assert each of john and paul is a man.
<19>  assert each of jane and pam is a woman.
<20>  assert any man is a person and any woman is a person.
<21>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.
```

```
    ANY MAN is a person.
    ANY WOMAN is a person.

<22>  display every person.
JOHN
PAUL
JANE
PAM
```

The **clear database** action deletes all sentences from the database. Lines <18>, <19>, <20>, and <21> are all of a form discussed above.

```
<23>  deny any man is a person and any woman is a person.
<24>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.

<25>  assert every man is a person and every woman is a person.
<26>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.
    JOHN is a person.
    PAUL is a person.
    JANE is a person.
    PAM is a person.
```

In line <23> we undo the assertion of line <20> and in line <25> we assert what is, in some sense, its intuitive equivalent. But the database displayed by line <26> is very different from that displayed by line <21>. This is because **every**, unlike **any**, does not delay evaluation. Rather, it immediately generates the same relationships that **any** would generate when called upon to do so. You can see that the real trade-off between **any** and **every** is one of time versus space. Using **any** saves you space but causes recomputation every time the relation is referenced.

Note, however, that if you add "Mary is a woman" to the database displayed by line <26>, the relationship "Mary is a person" would not appear in the database.

```
<27>  forget about person.
<28>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.
    JOHN is a person.
    PAUL is a person.
    JANE is a person.
    PAM is a person.

<29>  deny every man is a person and every woman is a person.
<30>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.
```

In <27> we attempt to clean out the database the way we did in <12> above. This time, however, the database remains unchanged, because we supplied the **forget** action with a relation-name instead of an element. This may be confusing at first because we are used to thinking about all nouns as being the same sort of thing. An easy way to tell if something in the database is an element or not is by whether it is capitalized or not. In <29> we use the correct form to effect the desired change.

```
<31>  load demoprog.
To decide INDIVIDUAL is a person
To generate PERSON
To decide INDIVIDUAL is married
```

All of the actions you have seen so far are single rules submitted to ROSIE's "top level." As in any high-level programming language, rules can be gathered together to

form purposeful units. In ROSIE these units are called "rulesets" and they come in several flavors. Two types of rulesets, generators and predicates, were previously written into a file called "demoprog" using the **build** and **edit** actions. When the editing was finished, ROSIE parsed the file and left demoprog.parse in the current directory. The **load** action looks for that parse file (or a compile file if it exists) in order to load it. ROSIE reminds you what is in the file as it is loaded.

```
<32>  scan demoprog.

To decide INDIVIDUAL IS A PERSON
    File DEMOPROG, 2 rules, NOT compiled.

To generate PERSON
    File DEMOPROG, 2 rules, NOT compiled.

To decide INDIVIDUAL IS MARRIED
    File DEMOPROG, 2 rules, NOT compiled.
```

Another way to find out the rulesets in a file is to use the **scan** action. Notice that this gives you only information about the rulesets in the file. To see the actual ruleset you use the **show** action (or edit the file) as in lines <33>, <34>, and <35> below.

```
<33>  show "To decide individual is a person".

To decide individual is a person:

[1] If the individual is a man or the individual is a woman
        conclude true.

[2] Conclude false.

End.
```

The ruleset definition above consists of three parts: a legal predicate header (To decide individual is a person.), the rules that make up the body of the ruleset

([1] and [2]), and the "endblock" (End) that must end every ruleset. Once defined, every time ROSIE tests "x is a person", this ruleset may be invoked, and whatever occurs in the x-position will be bound to "the individual". Notice that you must use the article **the** to refer to the bound value of the argument.

When a ruleset is invoked, a "private" database is set up for any ruleset arguments and private relationships. If you invoked this predicate by "if john is a person, display YES.", the relation is-an-individual(john) would be inserted into the ruleset's private database. Thereafter, any reference to "the individual" retrieves the one and only element satisfying the is-an-individual relation (i.e., the private database is always searched first). When the predicate terminates, the private database is discarded.

Predicates use the **conclude** action to return a value of "true" or "false". Because ROSIE uses a three-valued logic, a predicate can also use the **return** action to indicate that there is no element in the database that satisfies the predicate or the predicate's negation. Notice also that the "if" construction above does not require the word "then" as is usual in most high-level programming languages.

```
<34>  show "To generate person".

To generate person:

[1] Produce every man.

[2] Produce every woman.

End.
```

A generator ruleset produces the elements satisfying a given description. It does this by executing the rules that make up the body of the generator. Even if the rules of the generator produce a particular element more than once, ROSIE insures that that element is produced by the generator only once. The number of actual elements a generator produces depends upon how it is invoked. If we say "display every person," all elements

produced by the rules in the generator body will be displayed. If we say "display a person", the generator will produce only one person. Notice how generators and predicates work in tandem; generators produce the members of a class and predicates test a particular element for class membership.

```
<35>  show "To decide individual is married".

To decide individual is married:

[1] If the individual = pam or the individual = paul
        conclude true.

[2] Conclude false.

End.
```

This predicate is constructed similarly to the one above. It illustrates the fact that ROSIE permits the definition of semantic concepts such as "individual is married" to the depth needed to solve the programming task at hand. The most general definition is not needed! Predicates and generators are only two of the three types of rulesets. A third type is called a procedure and is used to collect together rules that perform a particular task. The procedure is invoked by using the **go** or **call** action. ROSIE procedures are quite similar to procedures in other high-level languages. Procedures are discussed at length in section 5.4 of this manual.

```
<36>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.

<37>  display every person.
JOHN
PAUL
```

```
JANE
PAM
```

Here we have an example of generator invocation. Notice that there is no apparent difference between lines <37> and <22>, although the former is a generator call and the latter is a reference to a class element. As far as the user is concerned, class elements and generators are indistinguishable.

```
<38>  if john is a person display YES.
YES
<39>  if pam is married display YES.
YES
```

In lines <38> and <39> we have examples of predicate invocation. Notice again that there is no discernible difference between the use of the predicate and that of either a class element or the explicit appearance of the relationships in the database. In other words, we could replace the predicate "is a person" with "assert any man is a person and any woman is a person.", or with "assert each of john, paul, jane and pam is a person."

```
<40>  assert Jane is married.
<41>  ?
[ Global Database ]
    JOHN is a man.
    PAUL is a man.
    JANE is a woman.
    PAM is a woman.
    JANE is married.

<42>  display every person who is married.
JANE
PAUL
PAM
```

Although in our discussion of lines <38> and <39> we noted that the use of class elements, predicates, and explicit relations were equivalent, there is one aspect of rulesets of which the user should be aware. In <42> we ask for every person who is married; clearly, Jane is married, since this fact is represented explicitly in the

database. The database relation "is-married" is accessed first; then the predicate is invoked.

```
<43>  assert john is not a man.
<44>  ?
[ Global Database ]
    PAUL is a man.
    JOHN is not a man.
    JANE is a woman.
    PAM is a woman.
    JANE is married.
```

In lines <8> and <12> above we saw two different ways of deleting information from the database. Line <43> shows us the third. Since ROSIE does not allow explicit contradictions in the database, asserting the negation of a relationship that already exists in the database automatically removes that relationship and adds its negation. Thus, in the example above, "JOHN is a man" is denied and "JOHN is not a man" is asserted.

```
<45>  deny Jane is married and assert Jane is engaged to tony.
<46>  assert tony is engaged and tony is a man.
<47>  ?
[ Global Database ]
    PAUL is a man.
    JOHN is not a man.
    TONY is a man.
    JANE is a woman.
    PAM is a woman.
    JANE is engaged to TONY.
    TONY is engaged.

<48>  display every person who is engaged.
TONY
<49>  display every person who is engaged to tony.
JANE
```

In lines <45> and <46> we add some new information to the database. In line <49>, however, it appears that ROSIE is mistaken. Surely, if Jane is engaged to Tony, Jane is engaged. Fortunately or unfortunately, this is not the case in ROSIE. In "JANE is engaged to TONY.", the relation is-engaged-to is dyadic, i.e., takes two

arguments (JANE and TONY). In the sentence "TONY is engaged.", the relation is-engaged is monadic, i.e., has only one argument (TONY). Thus, when we ask for "every person who is engaged" we are looking for those elements of the class defined by the relation is-engaged. In <49> we can get at the fact that Jane is also engaged by using the correct relation. Note that the arguments in the dyadic relation are ordered. By this we mean that if we say "display every person who is engaged to any person.", we would still only find Jane listed; just because Jane is engaged to Tony, ROSIE does not assume Tony is engaged to Jane.

```
<50>  deny tony is a man and assert tony is a person.
<51>  display every person.
TONY
PAUL
JANE
PAM
```

As in the case of predicates, the database is searched before the generator is invoked.

```
<52>  if there is a person who is engaged to tony, assert that
          person is engaged and assert tony is a man.
<53>  ?
[ Global Database ]
    PAUL is a man.
    JOHN is not a man.
    TONY is a man.
    JANE is a woman.
    PAM is a woman.
    TONY is a person.
    JANE is engaged to TONY.
    TONY is engaged.
    JANE is engaged.
```

Line <52> is a common form for a ROSIE rule: **if** condition actionblock. Notice the use of "that person" to refer to the person who is engaged to tony. Note also the use of the **and** to accomplish more than

one action. Line <52> gives you some idea of the complexity ROSIE rules can have.

```
<54>  for each person who is engaged
      send {that person," is engaged.",return}.
TONY is engaged.
JANE is engaged.
```

Line <54> presents a general format for output: the **send** action. **Send** takes a pattern as its "argument". A pattern, in turn, is made up of subpatterns. These subpatterns may consist of a term ("that person") which evaluates to an element before it is printed, a literal string (" is engaged."), or special subpatterns that have predefined meanings (return). The **send** action can be used to write to a file, another computer port, or the user's terminal (the terminal is the default argument as displayed above).

```
<55>  tony ?
[ TONY ]
    TONY is a man.
    TONY is a person.
    JANE is engaged to TONY.
    TONY is engaged.

<56>  describe tony.
[ TONY ]
    TONY is a man.
    TONY is a person.
    JANE is engaged to TONY.
    TONY is engaged.
```

Databases can grow to be quite large, and you do not always want to have to read the whole database to see if a particular set of relationships is present. In particular you may want to see all the relations involving a particular element; <55> and <56> show equivalent actions for accomplishing this.

```
<57>  ??

<18> ASSERT EACH OF JOHN AND PAUL IS A MAN.
<19> ASSERT EACH OF JANE AND PAM IS A WOMAN.
<20> ASSERT ANY MAN IS A PERSON AND ANY WOMAN IS A PERSON.
<21> ?
<22> DISPLAY EVERY PERSON.
<23> DENY ANY MAN IS A PERSON AND ANY WOMAN IS A PERSON.
<24> ?
<25> ASSERT EVERY MAN IS A PERSON AND EVERY WOMAN IS A PERSON.
<26> ?
<27> FORGET ABOUT PERSON.
<28> ?
<29> DENY EVERY MAN IS A PERSON AND EVERY WOMAN IS A PERSON.
<30> ?
<31> LOAD DEMOPROG.
<32> SCAN DEMOPROG.
<33> SHOW "TO DECIDE INDIVIDUAL IS A PERSON".
<34> SHOW "TO GENERATE PERSON".
<35> SHOW "TO DECIDE INDIVIDUAL IS MARRIED".
<36> ?
<37> DISPLAY EVERY PERSON.
<38> IF JOHN IS A PERSON DISPLAY YES.
<39> IF PAM IS MARRIED DISPLAY YES.
<40> ASSERT JANE IS MARRIED.
<41> ?
<42> DISPLAY EVERY PERSON WHO IS MARRIED.
<43> ASSERT JOHN IS NOT A MAN.
<44> ?
<45> DENY JANE IS MARRIED AND ASSERT JANE IS ENGAGED TO TONY.
<46> ASSERT TONY IS ENGAGED AND TONY IS A MAN.
<47> ?
<48> DISPLAY EVERY PERSON WHO IS ENGAGED.
<49> DISPLAY EVERY PERSON WHO IS ENGAGED TO TONY.
<50> DENY TONY IS A MAN AND ASSERT TONY IS A PERSON.
<51> DISPLAY EVERY PERSON.
<52> IF THERE IS A PERSON WHO IS ENGAGED TO TONY, ASSERT THAT
     PERSON IS ENGAGED AND ASSERT TONY IS A MAN.
<53> ?
<54> FOR EACH PERSON WHO IS ENGAGED SEND {THAT PERSON, " is
     engaged.", RETURN}.
<55> TONY?
<56> DESCRIBE TONY.
<57> ??
```

ROSIE keeps track of the last forty lines typed. To see all forty, type "??" as in <57>. Once you know the rule

associated with a line number, you can refer to that line number. This allows you to **redo** that rule as in <58>, below, or edit that rule.

```
<58>  redo 28.
[ Global Database ]
    PAUL is a man.
    JOHN is not a man.
    TONY is a man.
    JANE is a woman.
    PAM is a woman.
    TONY is a person.
    JANE is engaged to TONY.
    TONY is engaged.
    JANE is engaged.
```

Notice that even though we are redoing the ? action of line <28>, the database displayed is the current one.

```
<59>  logout.
```

End the ROSIE session. Note that the **logout** action automatically ends the **dribble** begun in line <1> and closes the file being "dribbled" to.

APPENDIX B

A ROSIE PRIMER

This primer familiarizes the new user with some of the more basic ROSIE concepts. These concepts should enable the newcomer to utilize the Reference Manual more effectively and, in addition, should dispel some of the initial confusion associated with learning a new system.

The primer consists of three sections. The first section is a glossary of terms. The second section gives an overview of ROSIE as a system. Here the interaction of the top level, the database, and the filepackage are explained. Finally, the third section provides an explanation of the most frequently encountered "actions." These three sections should provide the reader with a firm base for exploring ROSIE.

B.1. GLOSSARY

Definitions may use concepts defined elsewhere in the glossary. Such a word will appear in boldface.

ACTION

Some type of manipulation of data. An action may send output to the user's terminal (e.g., "display YES"), query the **database** (e.g,. "if john is a man, display YES"), invoke procedures (e.g., "go ask Mary"), affect the program **control structure** (e.g., "while john is a man, display YES"), etc. **Sentences** are the bases of actions and actions are the bases of **rules**.

ACTIONBLOCK

Any collection of **actions** joined by the word "and".

ALTERNATE DATABASE

Any **database** other than the **global database** or a ruleset's **private database**. An alternate database can be used to segregate information (i.e., hide it from the global database and other alternate databases).

BNF

The Backus-Naur form used to describe a language. The form consists of "productions" whose format is "<non-terminal> ::= some combination of terminals and <non-terminals>" where a non-terminal on the right-hand side

of the "::=" is always found on the left-hand side of another production and where a terminal is an actual word in the language.

BINDING

The process of associating a value to a **variable**. If we say "for each man (x) display x", then each **element** in the **database** that satisfies the relation is-a-man will in turn be bound to x. Once bound, the value of x in "display x" is the bound element.

CLASS ELEMENT

An **element** that stands for a class of elements. Represented using "any" followed by a **name**. Thus, "any man" refers simultaneously to the class of elements in the **database** satisfying the relation is-a-man.

COLONBLOCK

An **actionblock** optionally terminated with a semicolon. Used by the "select", "choose", and "match" actions.

COMMABLOCK

An **actionblock** optionally terminated with a comma. Used in place of parentheses whenever possible to increase readability.

CONDITION

Constructs that ask questions of the **database**. For example, "if john is a man" asks the database if the relation is-a-man exists for the **element** john.

CONTROL STRUCTURE

Actions that tell a program what to do and when. Looping, if-then statements, and case statements are the usual forms found in high-level programming languages. ROSIE has its own version of these forms.

DATABASE

The collection of asserted relations.

DESCRIPTION

A representation of a class of **elements**. Descriptions are used within **terms**, **conditions**, and **actions** to test for class membership or search for all the members of a class in the **database**. Descriptions may incorporate adjectives, relative clauses, prepositional phrases, and combinations of these.

ELEMENT

A primitive data type. Elements come in 6 flavors: **names**, **numbers**, **strings**, **tuples**, **propositions**, and **class elements**.

EVALUATION NAME

The **element** resulting when a **term** is evaluated. Thus, the evaluation name of the term "the teacher" might evaluate to the element "MARY SMITH" and the evaluation name of the term "3 + 7" is "10".

FILE

ROSIE maintains four files associated with the user's **program file**. The .TEXT file, .PARSE file, .MAP file, and .COMPILE file. These are explained further in connection with the **filepackage**.

FILESEGMENT

A **rule**, sequence of rules, or a **ruleset** within a .TEXT file. The file itself can also be considered a filesegment. ROSIE's **filepackage** allows you to work on single filesegments. This means ROSIE does not have to reparse every rule and ruleset in a file when one rule or ruleset is changed.

FILEPACKAGE

The filepackage refers to the three or four files associated with a user's **program file**. The .TEXT file contains the actual program listing in a readable form with rule numbers and uniform spacing added. The .PARSE file contains the executable parse of the items in the .TEXT file. The .MAP file contains information about the layout of the .TEXT, .PARSE, and .COMPILE, files thereby allowing the filepackage actions to access and change portions of a file as needed. The .COMPILE file (when it exists) contains the compiled version of the program file. The "load" action will load either the .PARSE file or the .COMPILE file, depending upon which has the newest creation date. A file must be loaded to permit editing or examination.

GENERATOR

A **ruleset** that generates elements. It is invoked via a **description**.

GLOBAL DATABASE

The global database is the default ROSIE database. **Alternate databases** can be created by the user to segregate information and make it inaccessible to the global database.

NAME

An **element** type. Name elements represent literal names and can contain more than one word separated by spaces. They cannot include words that can be interpreted as **numbers** or **strings**.

NAME TERM

A **term** that evaluates to a **name**.

NUMBER

An **element** type that represents a numeric value and the units or label associated with that value (if any). Numbers can be integer, floating point, or octal with optional positive or negative exponents.

NUMBER TERM

A **term** that evaluates to a **number**.

PATTERN

Constructs which allow the creation and manipulation of strings of text. A pattern is a sequence of subpatterns separated by commas and enclosed in braces. Each subpattern places a restriction on the string to be matched or generated. Example: {"John Doe", 2 blanks, "$12."} would match John Doe $12.

PREDICATE

A type of **ruleset** that is used to determine the truth or falsehood of relationships among **elements**. Predicates use the "conclude" action to return a value of true if the necessary relation can be found, or false if its negation is present in the **database**. A predicate can also use a simple "return" if neither the relation nor its negation can be found, thus giving ROSIE a three-valued logic.

PRIMITIVE SENTENCE

A sentence that determines a single relationship. There are five basic forms which can be extended by changes in tense and negation. They are:

term is a[n] **relation-name** optional-prep-phrase
term is **relation-name** optional-prep-phrase
term is **relation-name** **term** optional-prep-phrase
term does **relation-name** optional-prep-phrase
term does **relation-name** **term** optional-prep-phrase

PRIVATE DATABASE

The **database** associated with a **ruleset** when it is invoked. It is used to store intermediate results via private relations and disappears when the ruleset concludes execution.

PROCEDURE

A type of **ruleset**. Procedures are used for collecting together **rules** that perform a particular task. Procedures cannot return values.

PROGRAM FILE

The file the user builds and edits when writing ROSIE programs. The **filepackage** uses the program file to develop its own set of related **files**. The files produced by the filepackage provide ROSIE with an efficient means for examining and manipulating rulesets.

PROPOSITION

A **primitive sentence** enclosed in single quotes.

PSEUDO-TERM

The phrases "some", "every", "each of", and "one of" have a special meaning in ROSIE. They are **terms** that can evaluate to more than one **element**. This provides a unique way of scanning a group of elements to test a condition or perform an action. Example: "Assert each of John and Sue is a person." will add the relations is-a-person(John) and is-a-person(Sue) to the the **database**.

RELATION-NAME

The word provided by the user that helps distinguish the primitive relational forms. Thus, in the sentences "John is a man" and "John is a boy", "man" and "boy" are both relation-names. The relations themselves become is-a-man and is-a-boy.

RULE

One or more **actions** connected by the word "and" and terminated by the character "." or "!". Rules enable the user to manipulate the information in the **database**.

RULESET

A collection of **rules** that embodies a type of procedural knowledge. The three types of rulesets are **procedures**, **generators**, and **predicates**. Rulesets cannot be defined at the top level of ROSIE; they must be written into a **program file** which is then parsed and loaded.

STRING

An **element** that represents any number of characters found between double quotes.

STRING TERM

A **term** that evaluates to a **string**.

SYSTEM GENERATOR

A **generator** provided by the System Ruleset Library or written by the user in INTERLISP. A sample System Ruleset generator: "negation of element" returns the negation of a **number** or a **proposition**.

SYSTEM PREDICATE

A **predicate** provided by the System Ruleset Library or written by the user in INTERLISP. Sample: "element is a name" returns true if the element is a **name**; otherwise returns false.

SYSTEM PROCEDURE

A **procedure** provided by the System Ruleset Library or written by the user in INTERLISP. Sample: "show database" displays the contents of the **database** named.

SYSTEM RULESET

Predefined or user-written **rulesets** that perform useful operations. System rulesets are written in INTERLISP.

TERM

The syntactic construct that represents an **element**. The evaluation of a term produces an element. Thus, if "john's mother" is the term, then "mary smith" might be the element it represents.

TUPLE

An **element** that represents an ordered list of elements. Elements in the list are separated by commas, and the list itself is surrounded by left and right angle brackets. Example: <John, 5, "this string">.

TUPLE TERM

A **term** that evaluates into a **tuple** where the individual **elements** comprising the tuple have also been evaluated.

VARIABLE

A **name** used to stand for another **element**. Thus, in "for each person (x), display x", x is the variable that stands for each person produced by the "for each" action. We call the process of assigning a variable a value "**binding**".

VARIABLE TERM

A **term** that evaluates to the element bound to the variable.

B.2. THE ROSIE ENVIRONMENT

The ROSIE environment facilitates the creation of rule-based expert systems. It may be useful to think of ROSIE as a system comprised of three parts which work together but contain/demand different sorts of information. The three parts are the filepackage files, the database, and the top level. Their interrelationships are described below.

The TOP LEVEL: In TOPS-20, when you type "ROSIE" to the EXEC you will be placed at ROSIE's top level. At this point you are free to type any single rule to ROSIE. You can assert a sentence into the database, interface to TOPS-20 through "dir" and similar commands, interrogate or display the database, or load files. What you cannot do is define rulesets. Rulesets are aggregates of rules and are, in some sense, a higher-level information packet than you can construct in this part of the environment. You can load rulesets into this part of ROSIE, but even then they are accessible only through invocation from within a single rule.

The DATABASE: The top level allows you access to information in the database. The database is where the information expressed in an assertion or denial is stored. The information is stored in terms of relations between elements. You cannot enter the database the

way you can a file or the top level--your access to information at the database level is through the use of ROSIE actions that manipulate those relations in different ways. Thus, in some sense, the database is at a "lower" level than the top level, since it can only be accessed by going through the top level and since its representation of information is more primitive.

The FILEPACKAGE: When you write a program in ROSIE, the filepackage creates a set of files to facilitate changes to and examination of your program. Programs are where you can use rulesets--collections of rules that may define a concept (predicate rulesets), generate elements satisfying some description (generator rulesets), or pull together into one location a set of rules that perform a discrete task (procedure rulesets). Program files can also contain single rules. In this sense, the filepackage level is "higher" than the top level, since it allows you access to those types of information available at the top level as well as those that are not.

B.3. BASIC ACTIONS

The following list presents some actions that are frequently used in ROSIE programming. The form given is not always the most general form available but, rather, is the form that reflects a common usage.

DATABASE ACTIONS

assert primitive-sentence: adds the primitive-sentence to the database (e.g., assert mary does like tom's new cat).

deny primitive-sentence: removes the primitive-sentence from the database (e.g., deny paul does not need money).

let term be the description: removes from the database all relations for the given description and asserts term to be the element satisfying that description (e.g., let john be the boy with blond hair.).

forget about term: removes all relations involving the term from the database.

?: displays the sentences in the current database.

term ?: displays any sentence in the current database involving the term.

clear database: removes all relations in the current database.

activate term: makes term the current database.

INPUT and OUTPUT ACTIONS

display term: prints the evaluation name of the term on the user's terminal, followed by a carriage return.

open term to write: opens the directory file named term for writing. You can also open a file to read or append.

send to term pattern: creates a string from the pattern and sends it to the device referred to by term. If the phrase "to term" is omitted, the string is sent to the user's terminal.

read from term pattern: reads one character at a time from the device referred to by term. Will give an error if at any time the input precludes matching the pattern.

dribble to term: sends a copy of a ROSIE session to the file named by term. Sends everything that transpires in the session until the "logout" or "stop dribbling" action. (Actually, excursions into the editor are not recorded.)

PROGRAM CONTROL ACTIONS

if condition action: if the condition tests true, the action is performed.

for each description action: performs the action for each element satisfying the description.

while condition action: condition is tested before each iteration of the loop.

until condition action: condition is tested after each iteration of the loop.

go relation-name: invokes the procedure identified by relation-name.

return: terminates a ruleset. In a procedure it simply ends the ruleset; in a generator it causes the generator to stop generating elements; and in a predicate it causes the predicate to provide an indeterminate response.

produce term: used by generator rulesets to produce elements.
conclude true/false: used by predicate rulesets to indicate the truth value of the relation for a particular element.

logout: exits ROSIE to the EXEC.

FILEPACKAGE ACTIONS

load term: brings a program file into core. Causes ROSIE to "notice" any rules or rulesets so that they can later be examined or modified. Causes any ruleset without a syntax error to be "defined" so that it can be invoked.

build term: creates a new program file named term. Causes this file to be "noticed" so that you can edit it and thus create new rulesets.

parse term: used only when a program file is not created using the build action. In all other cases, return from the editor automatically invokes the parser to parse any new or modified rulesets.

show fileterm: displays the filesegment identified by fileterm to the terminal.

edit fileterm: allows you to edit a particular rule, ruleset, or file identified by fileterm.

scan fileterm: outlines the contents of the file named by fileterm. Outline includes naming rulesets, indicating compile status, pinpointing syntax errors, etc.

trace fileterm: a debugging aid that allows you to monitor the invocation of one or more rulesets. Typing simply "trace." at the top level will trace all currently defined rulesets. Using "trace fileterm" where fileterm is a file name will trace all the rulesets associated with that file. "Trace fileterm" may also be used with a single ruleset name. You cannot trace a single rule. The "untrace" action works in an analogous manner.

USER SUPPORT ACTIONS

??: displays the last forty lines of the current ROSIE session.

redo linenumber: allows you to resubmit one of the last forty lines.

fix linenumber: allows you to edit and resubmit one of the last forty lines.

info files: enumerates the filepackage files in your directory and whether or not they are loaded.

info loaded: tells which files have been loaded.

info storage: calls the garbage collector, then prints a detailed record of memory space. The bottom of the report tells how many pages of memory are free.

APPENDIX C

SYSTEM SUPPORT LIBRARY

[SYSTEM RULESETS]

```
SYSTEM RULESET TO DECIDE ELT IS A THING:
(NLAMBDA (ELT) '<TRUE>)


SYSTEM RULESET TO DECIDE ELT IS A PROPOSITION:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'PROPOSITION) THEN '<TRUE>
                ELSE '<FALSE>))


SYSTEM RULESET TO DECIDE ELT IS A TUPLE:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'TUPLE) THEN '<TRUE>
                ELSE '<FALSE>))


SYSTEM RULESET TO DECIDE ELT IS A STRING:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'STRING) THEN '<TRUE>
                ELSE '<FALSE>))


SYSTEM RULESET TO DECIDE ELT IS A NAME:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'NAME) THEN '<TRUE>
                ELSE '<FALSE>))


SYSTEM RULESET TO DECIDE ELT IS A NUMBER:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'NUMBER) THEN '<TRUE>
                ELSE '<FALSE>))


SYSTEM RULESET TO DECIDE ELT IS A CLASS:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'CLASS) THEN '<TRUE>
                ELSE '<FALSE>))


SYSTEM RULESET TO DECIDE ELT IS A FILESEGMENT:
(NLAMBDA (ELT) (IF (EQ (ELTTYPE ELT) 'FILESEGMENT) THEN '<TRUE>
                ELSE '<FALSE>))
```

```
SYSTEM RULESET TO GENERATE ELEMENT_TYPE OF ELT:
(NLAMBDA (ELT) (ELTTYPE ELT))


SYSTEM RULESET TO GENERATE INTEGER FROM INT1 TO INT2:
(NLAMBDA (INT1 INT2)
        (IF (NULL (FIXP INT1)) THEN
                (ABORT "Not an integer:" (ELTTOTOKENS INT1)))
        (IF (NULL (FIXP INT2)) THEN
                (ABORT "Not an integer:" (ELTTOTOKENS INT2)))
        (FOR I FROM INT1 TO INT2 COLLECT I))


SYSTEM RULESET TO GENERATE INTEGER FROM INT1 TO INT2 BY INT3:
(NLAMBDA (INT3 INT1 INT2)
        (IF (NULL (FIXP INT1)) THEN
                (ABORT "Not an integer:" (ELTTOTOKENS INT1)))
        (IF (NULL (FIXP INT2)) THEN
                (ABORT "Not an integer:" (ELTTOTOKENS INT2)))
        (IF (NULL (FIXP INT3)) THEN
                (ABORT "Not an integer:" (ELTTOTOKENS INT3)))
        (FOR I FROM INT1 TO INT2 BY INT3 COLLECT I))


SYSTEM RULESET TO GENERATE NUMBER_VALUE OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (GETNUMBERVAL NUM))


SYSTEM RULESET TO GENERATE NEGATION OF NUM/PROP:
(NLAMBDA (NUM/PROP)
         (IF (EQ (ELTTYPE NUM/PROP) 'NUMBER)
             THEN (IF (ATOM NUM/PROP)
                      THEN (MINUS (GETNUMBERVAL NUM/PROP))
                      ELSE (RPLACA (CDR NUM/PROP)
                                   (MINUS (GETNUMBERVAL NUM/PROP)))
                           (CONS NUM/PROP))
             ELSEIF (EQ (ELTTYPE NUM/PROP) 'PROPOSITION)
                 THEN (RPLACA (CDDDR NUM/PROP)
                              (IF (CADDDR NUM/PROP)
                                  THEN NIL
                                  ELSE T))
                      (CONS NUM/PROP)
                 ELSE (ABORT "Not negatable:" (ELTTOTOKENS NUM/PROP))))
```

```
SYSTEM RULESET TO GENERATE DATABASE:
(NLAMBDA ()
         (PROG (NAME)
           (COND [(SETQ NAME (GETPROP '<ACTIVATE> 'DB-NAME))
                  (COND [(EQ (CHCON1 NAME) 40)
                         (RETURN (CONS (CONS 'name
                                             (READ (MKSTRING NAME)))))]
                        [T (RETURN NAME)])]
                 [T (RETURN 'GLOBAL)])))


SYSTEM RULESET TO GENERATE DATABASES:
(NLAMBDA ()
         (CONS
          (CONS 'tuple
                (CONS 'GLOBAL
                      (for X in (GETPROP '<ACTIVATE> 'DATABASE-LIST)
                           collect (COND [(EQ (CHCON1 X) 40)
                                          (CONS 'name
                                                (READ (MKSTRING X)))]
                                         [T X])))))))


SYSTEM RULESET TO GENERATE UPPERCASE OF STRING:
(NLAMBDA (STRING)
        (IF (NEQ (ELTTYPE STRING) 'STRING) THEN
                (ABORT "Not a string:" (elttotokens string)))
        (U-CASE STRING))


SYSTEM RULESET TO GENERATE LOWERCASE OF STRING:
(NLAMBDA (STRING)
        (IF (NEQ (ELTTYPE STRING) 'STRING) THEN
                (ABORT "Not a string:" (elttotokens string)))
        (L-CASE STRING))


SYSTEM RULESET TO GENERATE SQUARE-ROOT OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (SQRT (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE SINE OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
```

```
        (SIN (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE SINE OF NUMBER IN RADIANS:
(NLAMBDA (RAD NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (SIN (GETNUMBERVAL NUM) T))


SYSTEM RULESET TO GENERATE ARCSINE OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ARCSIN (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE ARCSINE OF NUMBER IN RADIANS:
(NLAMBDA (RAD NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ARCSIN (GETNUMBERVAL NUM) T))


SYSTEM RULESET TO GENERATE COSINE OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (COS (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE COSINE OF NUMBER IN RADIANS:
(NLAMBDA (RAD NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (COS (GETNUMBERVAL NUM) T))


SYSTEM RULESET TO GENERATE ARCCOSINE OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ARCCOS (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE ARCCOSINE OF NUMBER IN RADIANS:
(NLAMBDA (RAD NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
```

```
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ARCCOS (GETNUMBERVAL NUM) T))


SYSTEM RULESET TO GENERATE TANGENT OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (TAN (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE TANGENT OF NUMBER IN RADIANS:
(NLAMBDA (RAD NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (TAN (GETNUMBERVAL NUM) T))


SYSTEM RULESET TO GENERATE ARCTANGENT OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ARCTAN (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE ARCTANGENT OF NUMBER IN RADIANS:
(NLAMBDA (RAD NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ARCTAN (GETNUMBERVAL NUM) T))


SYSTEM RULESET TO GENERATE FLOOR OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (FIX (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE LOG OF NUMBER:
(NLAMBDA (NUM)
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (LOG (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE ANTILOG OF NUMBER:
(NLAMBDA (NUM)
```

```
        (IF (NEQ (ELTTYPE NUM) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM)))
        (ANTILOG (GETNUMBERVAL NUM)))


SYSTEM RULESET TO GENERATE RANDOM_NUMBER FROM NUM1 TO NUM2:
(NLAMBDA (NUM1 NUM2)
        (IF (NEQ (ELTTYPE NUM1) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM1)))
        (IF (NEQ (ELTTYPE NUM2) 'NUMBER) THEN
                (ABORT "Not a number:" (ELTTOTOKENS NUM2)))
        (RAND (GETNUMBERVAL NUM1) (GETNUMBERVAL NUM2)))


SYSTEM RULESET TO GENERATE MEMBER OF TUPLE:
(NLAMBDA (TUPLE)
        (IF (NEQ (ELTTYPE TUPLE) 'TUPLE) THEN
                (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE)))
        (CDR TUPLE))


SYSTEM RULESET TO GENERATE MEMBER OF TUPLE AT INTEGER:
(NLAMBDA (INTEGER TUPLE)
        (IF (OR (NULL (FIXP INTEGER)) (ILEQ INTEGER 0)) THEN
                (ABORT "Not a positive integer:" (ELTTOTOKENS INTEGER)))
        (IF (NEQ (ELTTYPE TUPLE) 'TUPLE) THEN
                (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE)))
        (IF (SETQ TUPLE (CAR (FNTH (CDR TUPLE) INTEGER))) THEN
                (CONS TUPLE)))


SYSTEM RULESET TO GENERATE LENGTH OF TUPLE:
(NLAMBDA (TUPLE)
        (IF (NEQ (ELTTYPE TUPLE) 'TUPLE) THEN
                (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE)))
        (FLENGTH (CDR TUPLE)))


SYSTEM RULESET TO GENERATE TAIL OF TUPLE:
(NLAMBDA (TUPLE)
        (IF (NEQ (ELTTYPE TUPLE) 'TUPLE) THEN
                (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE)))
        (CONS (CONS (CAR TUPLE) (APPEND (CDDR TUPLE)))))


SYSTEM RULESET TO GENERATE TAIL OF TUPLE FROM INTEGER:
(NLAMBDA (INTEGER TUPLE)
        (IF (OR (NULL (FIXP INTEGER)) (ILEQ INTEGER 0)) THEN
```

```
            (ABORT "Not a positive integer:" (ELTTOTOKENS INTEGER)))
        (IF (NEQ (ELTTYPE TUPLE) 'TUPLE) THEN
            (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE)))
        (CONS (CONS (CAR TUPLE) (APPEND (FNTH (CDR TUPLE) INTEGER)))))
```

```
SYSTEM RULESET TO GENERATE CONCATENATION OF TUPLE1 WITH TUPLE2:
(NLAMBDA (TUPLE1 TUPLE2)
        (IF (NEQ (ELTTYPE TUPLE1) 'TUPLE) THEN
            (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE1)))
        (IF (NEQ (ELTTYPE TUPLE2) 'TUPLE) THEN
            (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE2)))
        (CONS (APPEND (CONS (CAR TUPLE1)) (CDR TUPLE1) (CDR TUPLE2) NIL)))
```

```
SYSTEM RULESET TO GENERATE REVERSE OF TUPLE:
(NLAMBDA (TUPLE)
        (IF (NEQ (ELTTYPE TUPLE) 'TUPLE) THEN
            (ABORT "Not a tuple:" (ELTTOTOKENS TUPLE)))
        (CONS (CONS (CAR TUPLE) (REVERSE (CDR TUPLE)))))
```

```
To decide PROP is true in  WORLD:

Private original-db.

[1] Let the original-db be the database.

[2] Activate the WORLD.

[3] If the PROP is provably true,
        activate the original-db and conclude true,
    otherwise activate the original-db and conclude false.

End.
```

```
To add PROP to WORLD:

Private original-db.

[1] Let the original-db be the database.

[2] Activate the WORLD.

[3] Assert the PROP is provably true.

[4] Activate the original-db.

End.
```

```
To remove PROP from WORLD:
Private original-db.
[1] Let the original-db be the database.
[2] Activate the WORLD.
[3] Deny the PROP is provably true.
[4] Activate the original-db.
End.

To show WORLD:
Private original-db.
[1] Let the original-db be the database.
[2] Activate the WORLD.
[3] ?
[4] Activate the original-db.
End.
```

REFERENCES

[1] Fain, J., Hayes-Roth, Sowizral, H., Waterman, D., Programming in ROSIE: An Introduction by Means of Examples. The Rand Corporation, N-1646-ARPA (forthcoming).

[2] Hayes-Roth, F., Gorlin, D., Rosenschein, S., Sowizral, H., Waterman, D., Rationale and Motivation for ROSIE. The Rand Corporation, N-1648-ARPA, November 1981.

INDEX